Automatic Program Restructuring for Distributed Memory Multicomputers

Mitsuru Ikei
B.S.E.E., Yokohama National University, 1982

A thesis submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science and Engineering

April 1992

The thesis "Automatic Program Restructuring for Distributed Memory Multicomputers" by Mitsuru Ikei has been examined and approved by the following Examination Committee:

Michael Wolfe
Associate Professor
Thesis Research Adviser

Steve W. Otto
Assistant Professor

Jingke Li
Portland State University
Assistant Professor

Acknowledgements

First I would like to thank my advisor, Michael Wolfe, for his support and encouragement of my research. Without his precise suggestions, I would not have been able to complete this work. I would also like to thank my thesis examiners, Jingke Li and Steve Otto, for their invaluable, inspiring comments and research discussion.

I would like to thank Hitachi Chemical Shimodate Research Laboratory President Keiji Hazama for giving me an opportunity to study here at Oregon Graduate Institute. I would also like to thank Senior Researcher Hiroyuki Iyama who taught me many research skills.

I would like to thank all my friends at OGI and Hitachi Chemical. At OGI, I would especially thank Harini Srinivasan who always spared time to discuss issues, and Jon Inouye for his help formatting the manuscript. I would also thank Miyoko Yagai of Hitachi Chemical who acted as a liason to Japan, and Atsushi Suzunaga who took care of the laboratory computers at Hitachi while I was working in the US.

Finally, I would like to thank my wife, Fujie and my children, Marie and Ray.

Contents

List of Tables

List of Figures

Abstract

Automatic Program Restructuring for
Distributed Memory Multicomputers

Mitsuru Ikei, M.S.
Oregon Graduate Institute of Science & Technology, 1992

Supervising Professor: Michael Wolfe

To compile a Single Program Multiple Data (SPMD) program for a Distributed Memory Multicomputer (DMMC), we need to find data that can be processed in parallel in the program and we need to distribute the data among processors such that the interprocessor communication becomes reasonably small. Loop restructuring is needed for finding parallelism in imperative programs and array alignment is one effective step to reduce interprocessor communication caused by array references. Automatic conversion of imperative programs using these two program restructuring steps has been implemented in the Tiny loop restructuring tool. The restructuring strategy is derived by translating the way that the compiler uses for the functional language Crystal, to the imperative language Tiny. Although an imperative language can have more varied loop structures than a functional language and it is more difficult to select the optimal one, we can get a loop structure which is comparable to Crystal. We also can find array alignment preference (temporal + spatial) relations in a Tiny source program and add a new construct, the align statement, to Tiny to express the array alignment preferences.

In this thesis, we discuss these program restructuring strategies which we used for Tiny by comparison with Crystal.

Chapter 1

Introduction

Scientific computer programs need more computation power and more memory. Distributed memory multicomputers (DMMC) can satisfy this need since the hardware is scalable both in computation size and memory size. Although many commercial DMMCs are available, software support for DMMC is currently insufficient. We need four kinds of additional software support to write a program which runs on a DMMC: (1)Process Creation, (2)Synchronization, (3)Data Distribution and (4)Communication. For each need of the software support, we can take two approaches, (a)the use of a new language construct which explicitly describe the manner of execution or (b)the use of a new construct or a new assumption (for an old construct) which implies some particular semantics.

Process Creation and Synchronization are necessary not only for DMMCs but also for the shared memory parallel computers (SMPC) such as CRAY X-MP, Alliant FX and Sequent Symmetry. We already have software support for this on SMPCs. These can be used for DMMCs. For Process Creation, the use of *fork* and *join* [1]is one solution, using approach (a). A programmer can explicitly specify the creation of a new processes by placing a *fork* and terminate the process by placing a *join* anywhere in the program. Although this pair of constructs is very powerful to express various kinds of concurrent execution patterns, it is also error-prone because it has too much freedom. The Single Program Multiple Data (SPMD) parallel programming model [Kar87] is another solution, but uses approach (b). In this model, one program is compiled and replicated on all

[1] *Join* is also a construct for synchronization.

processors. Each processor runs the same program but manipulates different parts of the data. All processes are created at the beginning of an execution and exist until the end of the program. In this thesis, we use SPMD since this model fits scientific applications.

Synchronization between some processors is needed when they want access to the same data. A *lock* is a construct for explicit synchronization which is used in PCF Parallel Fortran [Par91] . Our interest in this paper is only for synchronization which is implied by the loop constructs such as *PARALLEL DO* of PCF Parallel Fortran. Scientific application programs tend to have huge arrays and therefore we concentrate on data parallel algorithm [HS86]. Loops are the most common way to manipulate these arrays and often take most of the computing time, therefore we want to solve the synchronization problem in loops.

There are two types of language that support SPMD and implicit synchronization programming. These are:

- Language Extension: Parallel constructs are added to conventional an imperative language. Fortran 90[ISO91] and Fortran-D[FHK+90] are extensions to FORTRAN 77. Fortran 90 has array and vector manipulation extensions which implies parallel execution. Fortran-D has *FORALL* constructs to specify parallel execution explicitly.

- New Language: Design a new language for parallel programming. SISAL[OCA86] and Crystal[Li91] both belong to this category. Both are functional languages, so are declarative and free from side effects.

There are few languages which belong to one of above categories and can be compiled for DMMCs. Crystal has a compiler for DMMCs. Fortran-D has constructs to specify Data Distribution explicitly in contrast to Crystal and will depend on an aggressive compiler. Neither one has explicit communication constructs in their languages.

In this thesis, we use Language Extensions but we do not use explicit Data Distribution, unlike Fortran-D. We use Tiny[Wol90a] as a base language and try to achieve

implicit data decomposition on a conventional imperative language. The focus of this thesis is on Data Distribution. Although we assume communication can be implicit, automatic generation of communication primitives is left for future work. We use the way that Crystal compiler decomposes and distributes the data in their functional language, and apply it to an imperative language Tiny. Automatic generation of communication primitives should be similar to the method of Crystal compiler.

1.1 Distributed Memory Multicomputer

A DMMC is a computer which consists of many small processors. Each processor has local memory and bi-directional communication paths. All are connected by a communication network. Any two processors can send and receive data from each other, but there is no hardware support for a shared global memory space among the processors. In other words, each processor only has its own local address space.

There are several commercial DMMCs like iPSC/2, iPSC/860, Paragon, nCUBE 2 and CM-5 already on the market. Although no two of the above commercial machines have exactly the same processors and the same network topology, they have all adopted cut-through or worm-hole routing to send data to non directly-connected processors. This helps to keep communication time to be independent of the distance between two nodes by sending data in pipelining style.

To run an SPMD program on a DMMC, Data Distribution is needed. Since there is no global address space, it is obvious that large data such as huge arrays should be decomposed into pieces and distributed among all processors. Otherwise the program on a DMMC fails to be scalable in terms of memory size. To distribute the data, the following two properties should be considered carefully.

- Locality: Data access cost is not uniform. Data, that resides inside each processor can be referenced as cheaply as the memory access of a conventional sequential computer. Access to the data outside a processor needs a certain amount of time for communication.

- Communication Pattern: Communication can be slowed down by a communication contention between two or more pairs of processors. To avoid this contention, uniform or globally controlled communication is needed.

Data Distribution has a potential to cause communication since the necessary data can be placed outside of the processor which manipulates it. We want to stick the frequently accessed data in the processor which uses the data. Fortran 90 has array manipulation constructs such as *shift* and *spread* which implies uniform communication. These can be handled efficiently using the underlying communication network. Generating this kind of communication primitive is more feasible than generating only *send* and *receive* primitives. The goal of this thesis is to distribute large arrays in an SPMD program among processors such that (1)the communication between processors is reduced as much as possible and (2)the communication patterns become as uniform as possible and the communication can be executed efficiently by machine dependent communication libraries.

1.2 Compiling Steps

To run an SPMD program on DMMC, we need to convert the program to one which can be compiled (by a conventional compiler) for the target processor. The converted program will be linked to communication and other libraries, after the compilation. The input of this step is an SPMD user program which is written in an extended imperative language or a functional language. The output is the program which is written in an imperative language and calls interprocessor communication subroutines. We concentrate this conversion step in this thesis.

We divided this conversion into three phases (1)Loop Restructuring, (2)Array Alignment and (3)Communication Generation; see Figure 1.1. In the Loop Restructuring phase, a parallel program is transformed to a data parallel program. Since this output program does not have the information concerning data distribution, we call this a shared memory data parallel program. In the Array Alignment phase, all arrays in the

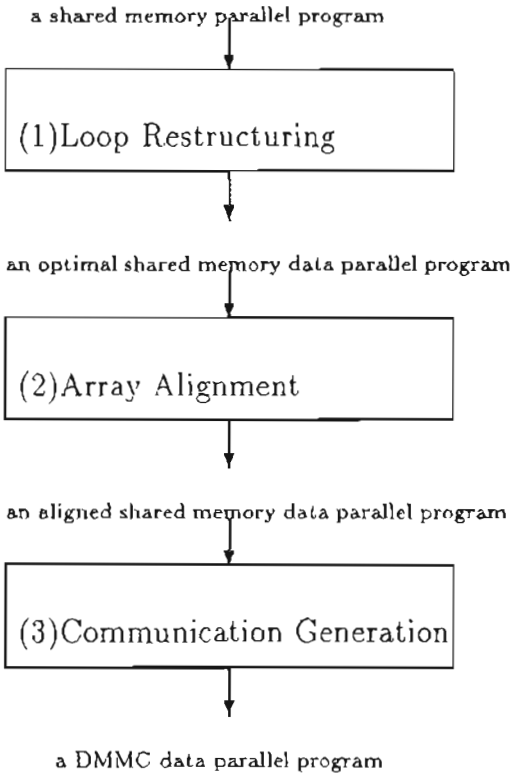


Figure 1.1: Compiling steps for DMMC

```
real a(1:100,1:100)
real b(1:100)
real c(1:100)
integer n

for i = 1,n do
  c(i) = 0.0
  doall j = 1,n do
    a(i,j) = a(i,j) + b(i)
  endfor
endfor
```

Figure 1.2: An example parallel program

program are aligned to some virtual processor array. In the Communication Generation phase, all arrays are actually distributed among physical processors and communication primitives are inserted in the program.

1.2.1 Loop Restructuring

In Figure 1.2, we show a small example parallel program which is written in the Tiny language. (This is not a realistic program. This is written only to show how compiling steps look like.) An input program looks like a usual sequential program except for the use of the parallel *doall* construct. This *doall* construct of Tiny has exactly the same semantics as *PARALLEL DO* of PCF Parallel Fortran. In this program, *b(i)* can be added to all members of the ith row of the array *a* in any order.

To convert this program to an optimal data parallel program, we need to select a loop that has uniform access to large data and can be executed parallel. Although explicitly described parallel loops are good candidates, sequential loops should also be considered to get better performance. Data dependence analysis is needed to detect parallelizable sequential loops.

In Figure 1.3, we show the output of this phase. The outer-most loop is parallelized and distributed as two independent, adjacent loops. All three loops becomes parallel

```
real a(1:100,1:100)
real b(1:100)
real c(1:100)
integer n

doall i = 1,n do
  c(i) = 0.0
endfor
doall i = 1,n do
  doall j = 1,n do
    a(i,j) = a(i,j) + b(i)
  endfor
endfor
```

Figure 1.3: An example optimal shared memory data parallel program

and selected as key loops for the data parallel program.

1.2.2 Array Alignment

Before decomposing all arrays to distribute them among processors, we want to relate all dimensions of arrays in the program to each other. This is done by mapping all arrays in a program to a unique virtual processor array. This phase is called array alignment. Fortran-D has an *align* construct for this purpose and the user has the responsibility for this. We added the same *align* construct to Tiny to specify the alignment of arrays in the intermediate output of this phase, although the *align* in Tiny is both syntactically and semantically slightly different from that of Fortran-D.

In Figure 1.4, we show the output of this phase. We use the largest (in terms of dimensionalities) array a for the virtual processor array. The relation between the largest array and the other arrays are declared in the *align* statements. The first *align* statement in the Figure 1.4 declares that array c should be aligned to the first dimension of array a. Only the processor which owns $a(i,1)$ should also own $c(i)$. In the second *align* statement, the array b is aligned similarly. Note that in both cases, array a is referenced as the virtual processor array.

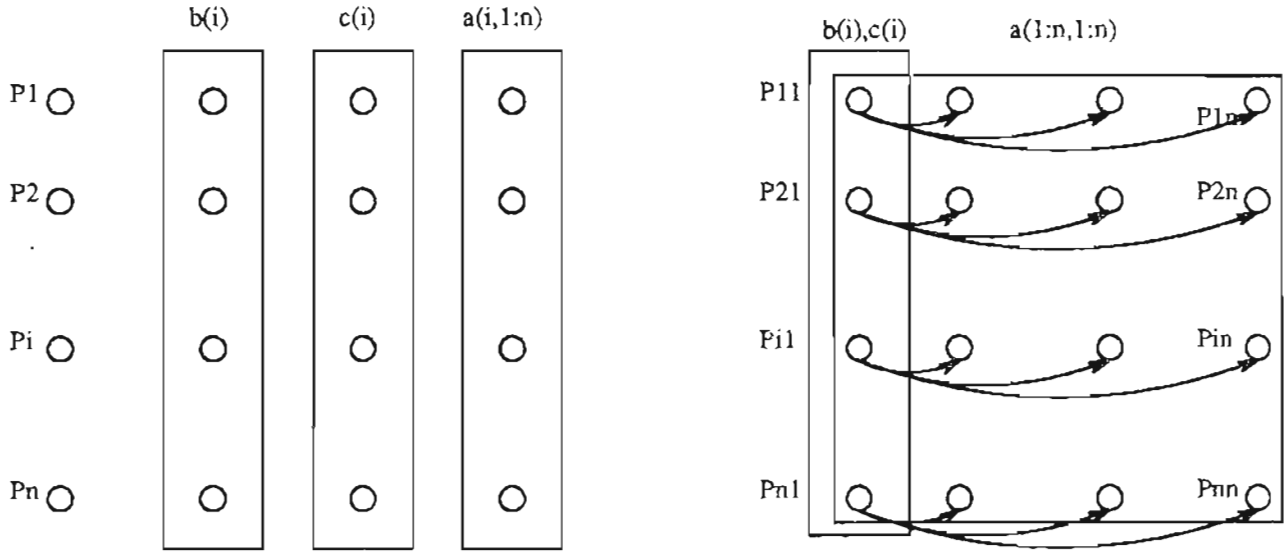
```
real a(1:100,1:100)
real b(1:100)
real c(1:100)
integer n

align(c(#1),a(#1,1))
align(b(#1),a(#1,1))

doall i = 1,n do
  c(i) = 0.0
endfor
doall i = 1,n do
  doall j = 1,n do
    a(i,j) = a(i,j) + b(i)
  endfor
endfor
```

Figure 1.4: An example aligned shared memory data parallel program

1.2.3 Communication Generation

Once the compiler has aligned all arrays in a program to some virtual processor array, the only array it needs to distribute is the virtual processor array since all other arrays are already mapped to it. There are many ways to distribute the virtual processor array to physical DMMC processors. In Figure 1.5, we show two examples of data distribution. For simplicity, here we assume that we have a vector of n processors or a square of $n \times n$ processors. In the 1-D case, the entire ith row of array a is stored in the single processor P_i. The ith elements of arrays b and c are also stored in P_i. In the other example, the *a(i,j)* is mapped to processor $P_{i,j}$. The ith elements of arrays b and c are stored only on $P_{i,1}$ by the constraints of *align* statement.

After data decomposition, each processor has two types of data which it uses and/or it defines. These are the data that the processor owns and the data that the other processors own. Each processor should receive the necessary data which belong to other processors before calculating and defining a new variable. Each processor should then send the updated data to any processors which use the new data.



(a) 1-D Distribution (b) 2-D Distribution

Figure 1.5: 1-D and 2-D array distribution of the example program

In Figure 1.5(a), there is no need for communication since all processors own all the data which they use. We show a DMMC program corresponding to Figure 1.5(b), in Figure 1.6 in quasi-Tiny language. We assume that the initial values of arrays are distributed before this program starts and the result will be gathered after the execution. Each processor has an unique processor id *proc_id* and the row and column positions can be calculated by the macros *my_row_pos* and *my_column_pos* respectively. In the body of first *if* statement, the processors which are in the first column set array c to 0.0. Similarly in the second *if-then-else* statement, the array b is multicasted row-wise from the processor $P_{i,1}$ and the others receive that value. The communication pattern of this program is shown in Figure 1.5.

Although some assumptions may not be practical in many cases, we can extend this method without losing generality. For instance, the assumption that we have n or $n \times n$ processors can be avoided by changing the DMMC program to calculate blocks of data instead of calculating one elements.

```
real a(1:100,1:100)
real b(1:100)
real c(1:100)
integer n

my_row = my_row_pos(proc_id)
my_column = my_column_pos(proc_id)

if(my_column == 1) then
  c(my_row) = 0.0
endif
if(my_column == 1) then
  multicast_in_row(b(my_row))
else
  receive(b(my_row))
endif
a(my_column,my_row) = a(my_column,my_row) + b(my_row)
```

Figure 1.6: A DMMC data parallel program corresponds to Figure 1.5(b)

1.3 Array Alignment and Communication

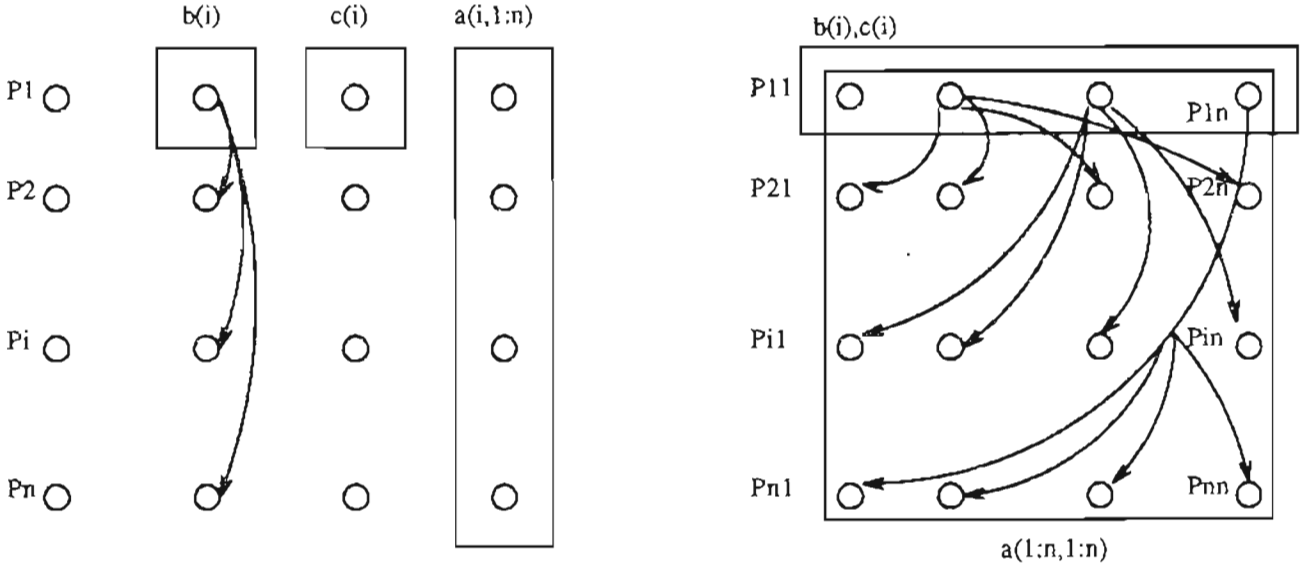
We showed that a shared memory parallel program can be converted to a SPMD program which runs on DMMCs through three steps. We used two steps, Array Alignment and Communication Generation, to distribute all arrays in a program to processor arrays. Array Alignment has an important role to reduce the communication between processors. Assume we have aligned as follows in the program of Figure 1.4.

align(c(#1),a(1,#1))

align(b(#1),a(1,#1))

Both array c and array b are aligned to the second dimension of array a. The distribution on physical processors becomes as shown in Figure 1.7.

In Figure 1.7(a), the entire array b and c are stored in processor P_1. Since all processors which have *a(i,j)* need *b(i)*, *b(i)* should be sent to processor P_i. Figure 1.7(b) is worse. Here, *b(i)* should be multicasted to the ith row of the processor array. It is



(a) 1-D Distribution

(b) 2-D Distribution

Figure 1.7: Misaligned array distribution of the example program

clear that the first alignment is better and can be executed efficiently.

This paper mainly discusses automatic array alignment. We want to derive an optimal aligned parallel program from an imperative source program which does not have explicit align statements. The rest of this paper is organized as follows. Chapter 2 discusses the compiling approach for the functional language Crystal; I used its strategy for array alignment of imperative programs. In Chapter 3 we discuss an imperative language Tiny, for which I implement automatic array alignment. Chapter 4 shows the differences between Tiny and Crystal, Chapter 5 discusses issues of program restructuring, Chapter 6 shows the algorithms and Chapter 7 is a conclusion.

Chapter 2

Crystal

Crystal is a functional language which has a compiler for DMMC. Crystal has neither parallel constructs nor data decomposition constructs. Even though everything is implicit, Crystal programs can be converted to iPSC/2 C programs by the steps in Figure 2.1. The first phase, Index Domain Alignment, is a necessary preprocess for the Control Structure phase. In this phase, all index domains of a Crystal program are aligned and the output becomes an aligned Crystal program. The three following steps almost correspond to the ones in Figure 1.1. In the Control Structure Synthesis phase, necessary control structures for execution are derived from the declarative description of Crystal. In the Spatial Domain Alignment phase, all index domains are re-aligned in the context of an imperative language. In the Communication Generation phase, communication primitives are inserted. In this section, we describe how the Crystal compiler converts programs in each of the four steps.

2.1 Program Example

In Figure 2.2, we show a Matrix Multiplication(MM) Crystal program as an example. This program consists of *index domain* definitions, the first four lines, and *data field* definitions, the rest of the program. An *index domain* definition defines the shape of composite data structure. An *index domain* can be either a basic index domain or a combination of basic domains which are combined by some operator. In this program, we use an interval domain *[0..n]* for T, and we use another interval domain *[1..n]* for $D1$.

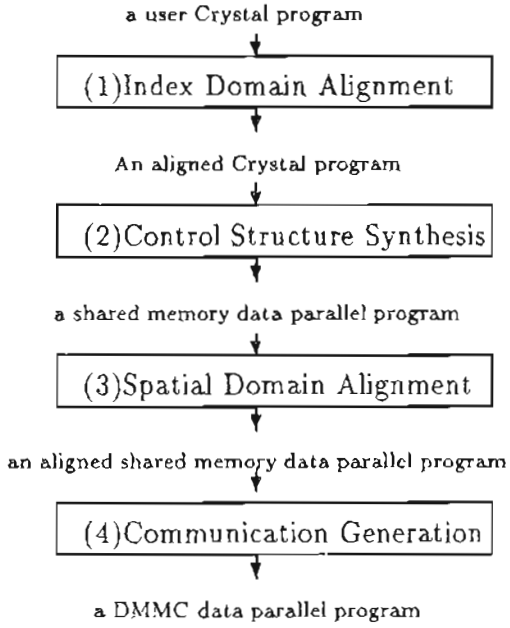


Figure 2.1: Crystal compiling steps for DMMC

```
dom T  = [0..n]
dom D1 = [1..n]
dom D2 = D1 * D1
dom D3 = D2 * T
dfield a(i,j): D2 = a0[i,j]
dfield b(i,j): D2 = b0[i,j]
dfield c(i,j,k): D3 = if (k = 0) then 0.0
                      || else c(i,j,k-1) + a(i,k) * b(k,j)
                      fi
```

Figure 2.2: An MM Crystal program

D2 is defined as a domain product of *D1* and *D1*. Similarly *D3* is defined as a product of *D2* and *T*.

A *data field* definition is a definition of a function over specified index domain. Both *a(i,j)* and *b(i,j)* are defined on *D2* to be initialized by the 2-D arrays *a0[i,j]* and *b0[i,j]*, respectively. In the definition of *c(i,j,k)* on *D3*, *c(i,j,0)* is set to 0.0 and *c(i,j,n)* has the result of the matrix multiplication. Crystal has a reduction operator "\" and the solution *c2(i,j)* could have been defined as:

```
dfield c2(i,j): D2 = \ + {a(i,k) * b(k,j) | 1<=k<=n}
```

Here we assume the program is written to assure the calculation order since later we want to compare the result to Tiny which has no reduction operator.

2.2 Index Domain Alignment

The first step to compile a Crystal program is Index Domain Alignment. The objective of this step is to align all *data fields* to the *data field* which has the highest dimensionality. The output program of this phase only has *data fields* which have the same shape, as shown in Figure 2.3. In this aligned version of Crystal program, all *data field* definitions are on *D3*. In the definition of *a(i,j,k)*, a new dimension is added as a second dimension *j* where all but *j=1* are set to 0.0. Similarly the new dimension *i* is added as the first dimension in the definition of *b(i,j,k)*.

We have two index domain alignment phases in Figure 2.1. Both these two phases consist of almost the same processes, (1)building a *component affinity graph*(CAG) and (2)finding the optimal partition of the CAG. Since the Spatial Domain Alignment phase is more closely related to the imperative language case, here we only show the result and the details are left for later in this chapter.

```
dom T  = [0..n]
dom D1 = [1..n]
dom D2 = D1 * D1
dom D3 = D1 * D1 * T
dfield a(i,j,k): D3 = if (j = 1) then a0[i,k]
                      || else 0.0
                      fi
dfield b(i,j,k): D3 = if (i = 1) then b0[k,j]
                      || else 0.0
                      fi
dfield c(i,j,k): D3 = if (k = 0) then 0.0
                      || else c(i,j,k-1) + a(i,1,k) * b(1,j,k)
                      fi
```

Figure 2.3: An aligned MM Crystal program

2.3 Control Structure Synthesis

In Control Structure Synthesis, an aligned Crystal program is converted to an imperative shared memory data parallel program. To compile a Crystal program, we need to derive at least one sequential order on which processors can execute its definition. We have to select the order such that the data dependence relations in the Data Field definition are preserved. To derive this order, the Crystal compiler takes two steps, (1)building a Call-Dependence Graph(CDG) and (2)deriving the loop structure from the CDG. We describe these two steps in this section.

2.3.1 Call-Dependence Graph

Call-Dependence Graph(CDG) is the graph which represents data dependences in a Crystal program. To build a CDG, first we make a list of *reference patterns*. *Reference patterns* are derived from *data field* definitions. A *reference pattern* has the following form:

$$a(i_1, \ldots, i_n) \leftarrow b(\tau_1, \ldots, \tau_n) \tag{2.1}$$

c(i,j,k) ← c(i,j,k-1)
c(i,j,k) ← a(i,1,k)
c(i,j,k) ← b(1,j,k)

Figure 2.4: Reference Patterns of MM program

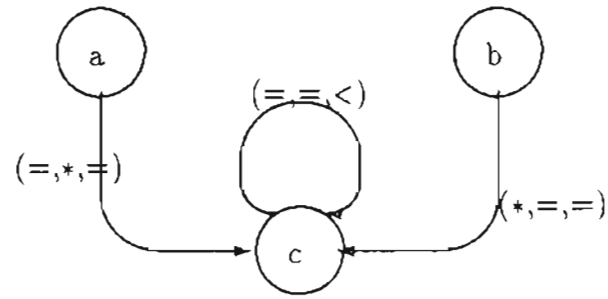


Figure 2.5: CDG of MM program

Above $a(i_1, \ldots, i_n)$ is the *data field* which is defined by a *data field* definition, and $b(\tau_1, \ldots, \tau_n)$ is used in the definition. This can be read $a(i_1, \ldots, i_n)$ depends on $b(\tau_1, \ldots, \tau_n)$ and the dependence is called *call-dependence*. In Figure 2.4, we show *reference patterns* of the example MM program in Figure 2.3. *Direction vectors*[Wol78, Wol89] can be defined on this *reference pattern* list in a similar way as originally defined. Using Form 2.1, here *direction vector* is defined as:

$$(sign(i_1 - \tau_1), \ldots, sign(i_n - \tau_n))$$

where

$$sign(\tau[i]) = \begin{cases} < & \text{if } \tau[i] < 0 \ \forall i \\ = & \text{if } \tau[i] = 0 \ \forall i \\ > & \text{if } \tau[i] > 0 \ \forall i \\ * & \text{otherwise} \end{cases}$$

From a *reference pattern* list, we can draw a CDG. The CDG is a directed graph such that each node represents a *data field* and each directed edge represents *call-dependence*. In Figure 2.5, we show a CDG of the example program in Figure 2.3. This graph is similar to a data dependence graph except for the following:

1. Dependence is defined between function definitions (*data fields*) not between statements(i.e. *call-dependence*)

2. There is neither anti-dependence nor output dependence.

In Figure 2.5, we also show *direction vectors* at corresponding edges.

2.3.2 Loop Derivation

Using the Call-Dependence Graph and *direction vectors*, we can derive a shared memory version of the parallel program. The Crystal compiler uses three kinds of loop constructs to form an execution order. These are:

for A sequential loop, which has the same semantics with *DO* in FORTRAN.

forall A parallel loop, which has the same semantics with *PARALLEL DO* in PCF Fortran.

while-active A special loop which can be executed by checking given *call-dependence* to handle difficult cyclic CDGs.

To derive necessary loop constructs, the Crystal compiler takes the following three strategies:

1. Treat each *strongly connected component* in CDG as a unit.

2. Apply topological sort to the *strongly connected components* of the CDG to decide the execution order of all units.

3. Break cycles in a unit by finding a *dependence carrying* component in the *direction vectors* involved in the cycle.

In Figure 2.5, the *strongly connected components* are $\{a\}$, $\{b\}$ and $\{c\}$. By topologically sorting the graph, execution order can be either *abc* or *bac*. Since $\{a\}$ and $\{b\}$ have no cycle, they both can form *forall* loops. To break a cycle in $\{c\}$, we have to form a *for*

```
dom T  = [0..n]
dom D1 = [1..n]
dom D2 = D1 * D1
dom D2T = D1 * T
dom D3 = D1 * D1 * T
forall ((i,k):D2T){
    a(i,k) = a0[i,k]
}
forall ((j,k):D2T){
    b(j,k) = b0[k,j]
}
for    (k: T){
    forall ((i,j):D2){
        c(i,j,k) = if (k = 0) then 0.0
                || else c(i,j,k-1) + a(i,k) * b(j,k)
                fi
    }
}
```

Figure 2.6: A SM data parallel MM program

loop for the *dependence carrying* component, k. The output of this phase is shown in Figure 2.6.

In this example, *strongly connected components* are all singletons and the cycle is a self cycle. Finding a *dependence carring component* was easy, so we didn't need to use *while-active* loops. In generally, we need *while-active* loops to resolve a cycle which has no *dependence carrying* components. We won't discuss this case since we don't use this kind of loop construct in imperative languages in this thesis. We also have a chance to get multiple *dependence carrying* loops. In this case, the Crystal compiler keeps multiple loop structures, which are ordered by the degrees of parallelism. The most effective one will be selected in a later phase. In Figure 2.6, the compiler applied minor optimizations to reduce necessary memory size. As the result, a and b return to be defined on 2-D domain.

$$c(i,j,k) \leftarrow c(i,j,k\text{-}1)$$
$$c(i,j,k) \leftarrow a(i,k)$$
$$c(i,j,k) \leftarrow b(j,k)$$

Figure 2.7: Reference Patterns of SM MM program

2.4 Spatial Domain Alignment

To reduce data movement during program execution, we want to align all *data fields* to a common *index domain* such that a related dimensions of *data field* definitions are the same dimension of the common *index domain*. This is exactly the same need which we had when we wanted to preprocess for Control Structure Synthesis. The only difference is that we are now interested in the dimensions of *data fields* which are distributed among processors because data movement along such dimensions has potential communication. We are not interested in the dimensions which are traversed sequentially, since such dimensions carry dependences and distributing along those dimensions doesn't make sense. These two domains are called the *spatial domain* and the *temporal domain*.

The Crystal compiler uses the following two steps, (1)building a *Component Affinity Graph*(CAG) and (2)finding the optimal partition of the CAG, to align a *spatial domain*. We describe these two steps in this section.

2.4.1 Component Affinity Graph

The CAG is weighted undirected graph. Nodes of CAG are *domain components* and edges are relations which are derived from the *reference patterns*. In Figure 2.7, we show *reference patterns* of an example program in Figure 2.6. We denote a *domain component* of a *reference pattern* as *dom(c,1)* which refers to the *1*-st domain component of the *data field c*. Given a *reference pattern*, we define a distance between a domain component in LHS and that in RHS as subtraction of an index expression in the RHS domain from the corresponding LHS index expression. If the distance between two different domain components *dom(p,i)* and *dom(q,j)* is constant, we say that there is a *affinity relation* between *dom(p,i)* and *dom(q,j)* and denote this as *(dom(p,i),dom(q,j)*. We can ignore the

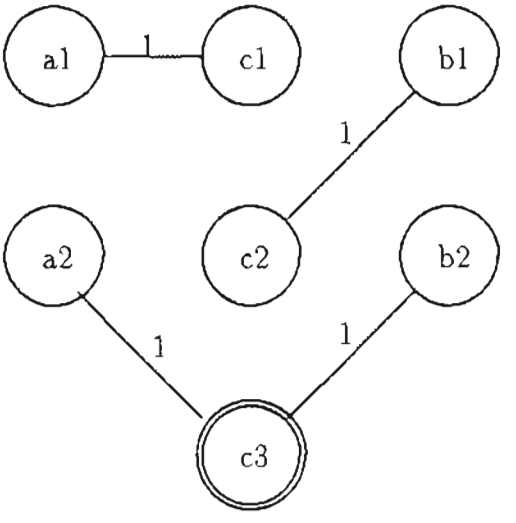


Figure 2.8: CAG of MM program

first *reference pattern* since we are only interested in inter-*data field* relations. From the second, we can derive two *affinity relations*, one between *domain component dom(c,1)*, which is the first component of *data field c*, and *dom(a,1)*, and other between *dom(c,3)* and *dom(a,2)*. These two *affinity relations* can be denoted by *(dom(c,1), dom(a,1))* and *(dom(c,3), dom(a,2))*. From the third *reference pattern*, two *affinity relations*, *(dom(c,2), dom(b,1))* and *(dom(c,3), dom(b,2))* are derived. In Figure 2.8, we show a CAG of the example program in Figure 2.6. The *domain components* of the same *data field* are place in the same column in this figure. There are two types of *domain components*, *spatial component* and *temporal component*. *Dom(c,3)* is the *temporal component* which is represented by double circles, since it is defined by the index k which resides in the *temporal domain*. All other components in the figure are *spatial components*.

The weights in the figure show strength or preference of these *affinity relations*. There are three levels of weight, ∞, 1 and ϵ. The 1 and ϵ shows strength of relation between two components. The ϵ represents a value much smaller than 1. If two relations derived from one *reference pattern* compete with each other, ϵs are assigned to these relations. For example, from the following reference pattern:

$$a(i,j) \leftarrow b(i,i)$$

two competing relations *(dom(a,1), dom(b,1))*, and *(dom(a,1), dom(b,2))* are derived and both edges will be rated as ϵ. Infinity shows preference rather than strength of relation. Edges between two *temporal components* are rated as ∞. Since we don't want to distribute data along a *temporal domain*, all *temporal components* would better be aligned to the same domain. In Figure 2.8, we have neither a competing relation nor a relation between two *temporal components*, so all edges are rated 1.

2.4.2 Optimal Partition of CAG

Using the CAG, the *spatial domain* alignment problem can be interpreted to the optimal partition problem of the CAG. We want to partition the CAG into n groups such that the sum of the edge weights cut by partitioning is minimal, assuming n is the maximum dimensionality of *index domains* in a program. Components which are derived from the same *data field* must be belong to different groups. This problem is NP-complete. The Crystal compiler uses heuristics to solve this problem. The rough sketch of this algorithm is:

1. Select one *data field*(column) which has the largest dimensionality as the common *index domain*.

2. For each *data field*(column) in CAG, form a bipartite graph with the common *index domain*. An edge is placed between two components in the bipartite graph if there is a path between the two components in CAG. The weight of the edge is the sum of all edges connected to the two nodes.

3. For each bipartite graph, align the target to the common *index domain* according to edge weights in the bipartite graph and reduce the CAG by the aligned *data field*.

In Figure 2.9, we show the partitioned CAG of the example MM program. Using the alignment in the figure, the aligned SM data parallel MM program can be derived as shown in Figure 2.10. In this program, new dimensions are added as the second

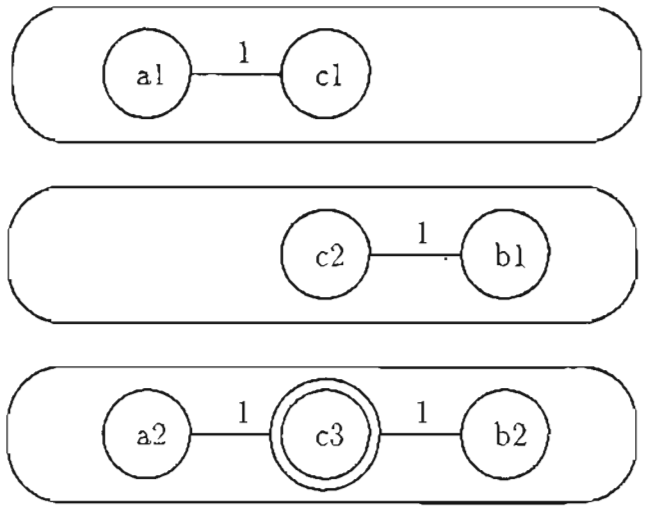


Figure 2.9: Partitioned CAG of MM program

```
dom T  = [0..n]
dom D1 = [1..n]
dom D2 = D1 * D1
dom D3 = D1 * D1 * T
forall ((i,j,k):D3){
    a(i,j,k) = if (j = 1) then a0[i,k]
               || else 0.0
               fi
}
forall ((i,j,k):D3){
    b(i,j,k) = if (i = 1) then b0[k,j]
               || else 0.0
               fi
}
for    (k: T){
    forall ((i,j):D2){
        c(i,j,k) = if (k = 0) then 0.0
                   || else c(i,j,k-1) + a(i,1,k) * b(1,j,k)
                   fi
    }
}
```

Figure 2.10: An aligned SM data parallel MM program

dimension j of *a(i,j,k)* and as the first dimension i of *b(i,j,k)* again. Since we don't have any ∞ edges in the CAG, the result of *spatial domain* alignment becomes exactly the same as the result of the previous *index domain* alignment.

2.5 Communication Generation

To run a program on DMMCs, we have to distribute data in the program among processors and insert communication primitives into the program. We know which *data fields* can be distributed from the aligned SM program. The *data field* which is defined by *forall* constructs can be distributed. Since we have aligned all *data fields* to the common *index domain*, the only *data field* we have to distribute is the one which was selected as the common *index domain*. However we still don't know which dimensions should be distributed nor which layout strategies(i.e. block, interleaving, etc.) should be taken to run the program efficiently. All these factors depend on the actual DMMC on which the program run. Not only the kind (architecture) of DMMC but also the configurations(i.e. numbers of processors, size of memory, etc.) of the computer should be considered at compile time. The Crystal compiler takes three steps for this phase. These are:

1. Distribute all *spatial domains* by blocked interleaving strategy. The number of processors and the block size are not fixed but given as parameters.

2. Generate communication primitives by pattern matching the *reference patterns* to the predefined communication primitives.

3. Select values of all the parameters by estimating computation and communication time of a given DMMC and optimize it further.

There are ten communication basic primitives: five *simple routines* and five *general routines*. (More primitives, such as gather and scatter can be added to this.) The only differences between these are that *simple routines* only communicate on a single dimension but *general routines* can communicate on multiple dimensions. In table 2.1, we show *simple routines* with their matching pattern. In the table, D is an *index domain*

Table 2.1: Simple communication routines and their matching patterns

Routines	*Patterns*
Spread(D, p, s, a, a_1, B)	$(i_1, .., i_p, .., i_n) \leftarrow (i_1 .., s_p, ..i_n)$
Reduce$(D, p, d, a, a_1, B, \oplus)$	$(i_1, .., d_p, .., i_n) \leftarrow (i_1 .., i_p, ..i_n)$
Multispread(D, p, a, a_1, B)	$(i_1, .., i_p, .., i_n) \leftarrow (i_1 .., j_p, ..i_n)$
Copy(D, p, s, d, a, a_1, B)	$(i_1, .., d_p, .., i_n) \leftarrow (i_1 .., s_p, ..i_n)$
Shift(D, p, c, a, a_1, B)	$(i_1, .., i_p + c, .., i_n) \leftarrow (i_1 .., i_p, ..i_n)$

representing a processor array, p is the dimension where they communicate, s denotes the index of the source processor, d denotes the destination processor index, a is a pointer to the input data, a_1 is a pointer to the output data, and B is the size of message. In the patterns, all subscript shows a position of indices, i_p and j_p are variable, d_p, s_p and c are constant.

Using these primitives, the Crystal compiler can derive the example DMMC data parallel program as shown in Figure 2.11. In this program, *index domain* E denotes a processor array which we use as a DMMC and *index domain* $D2B$ is the portion of global *data field* which each processor has. Since we take blocked interleaving distribution, we use *data field* P to describe how to interleave. For simplicity, we set E and $D2B$ as squares and we didn't define P. We can think of these as being parameterized. There are two macros used here. *Local_to_Global* can calculate the global index from P, a global index domain, a local index and a processor index. *Index_to_Pid* is used to find the processor index which has given global indices. The example MM program has been transformed to the program which explicitly communicates on a DMMC.

```
dom T  = [0..n]
dom D1 = [1..n]
dom D2 = D1 * D1
dom D3 = D1 * D1 * T
dom E  = [0..m][0..m]
dom D2B = [0..1][0..1]
forall ((p,q): E){
    forall ((il,jl):D2B){
        (i,j) = Local_to_Global(P,D2,(il,jl),(p,q))
        forall (k: T){
            al(il,k) = if (j = 1) then a0[i,k]
                       || else 0.0
                       fi
        }
    }
    forall ((il,jl):D2B){
        (i,j) = Local_to_Global(P,D2,(il,jl),(p,q))
        forall (k: T){
            bl(jl,k) = if (i = 1) then b0[k,j]
                       || else 0.0
                       fi
        }
    }
    for    (k: T){
        forall ((il,jl):D2B){
            (i,j) = Local_to_Global(P,D2,(il,jl),(p,q))
            (pi1,pj1) = Index_to_Pid(P,D2,(i,1))
            (pi2,pj2) = Index_to_Pid(P,D2,(1,j))
            Spread(E,2,(pi1,pj1),al(il,k),aa(il,k),1)
            Spread(E,1,(pi2,pj2),bl(jl,k),bb(jl,k),1)
            cl(il,jl,k) = if (k=0) then 0.0
                          || else cl(il,jl,k-1) + aa(il,k) * bb(jl,k)
                          fi
        }
    }
}
```

Figure 2.11: A DMMC data parallel MM program

Chapter 3

Tiny

Tiny is a program restructuring tool for imperative languages. The main objective of Tiny is loop restructuring. Tiny has a menu based user interface, which allows users to restructure loops interactively. During an interactive session, illegal restructurings(i.e. restructurings which may change the semantics of the original program) are detected by its built-in data dependence analysis, and users are informed automatically. After the session, Tiny can convert the program to a C program or a Fortran program with parallel execution directives. These programs can be compiled and run on Sequent or Alliant shared memory parallel computers(SMPC).

Tiny has its own imperative language to describe programs. (We use Tiny to mean both the tool and its language.) Unlike Fortran, this language is not designed to write big scientific programs. We can easily convert a well-formed loop structure as would be written by Fortran *DO* statements to a Tiny program and can restructure the program to get more parallelism, although it does not have several important features to write real application programs, like subroutine calls. We select Tiny as a base language for this project because:

1. Tiny is a simple imperative language with parallel loop constructs.

2. Tiny has strong loop restructuring features supported by its dependence analysis.

Our goal is to compile an imperative parallel program for distributed memory multicomputers(DMMC) by focusing on loop structures. The simplicity of Tiny allows us to

avoid other issues which require expensive interprocedural analysis [CK87]. We want to restructure a given parallel program to an appropriate data parallel form. Powerful restructuring features are feasible for this. In this chapter, we describe the features of Tiny which we use for this project.

3.1 Data Dependence

Since Tiny is an imperative language, a Tiny program does not show the definition of some function directly but does show the execution order of statements. In a Tiny program, functions are thought to be defined indirectly by the sequence of program statements. We want to restructure the program without changing the function which is coded into the program. However once a program is written as a sequence of executable statements, one can easily imagine that reproduction of the original function is very difficult. The most popular way to solve this problem is to restructure the program so that the effect of original (imperative) program is preserved. Data dependence analysis is the analysis of program execution effects in (the execution of) a sequence of statements. For instance, in the following program:

```
S1:a = b + c
S2:d = 2.
```

statement S_1 changes the value of a and statement S_2 changes the value of d. We can change the execution order of S_1 and S_2 since the values of a and d are independent of the execution order of these statements. There is no data dependence between S_1 and S_2. Note that all the statements independent each other can be executed any order, thus can be executed in parallel.

In the next example:

```
S1:a = b + c
S2:d = a + 2.
```

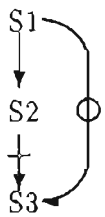
we can not change the execution order of S_1 and S_2 since S_2 uses a to calculate d and a is computed by S_1. In other words, S_2 depends on S_1. This data dependence is called *true dependence* or *flow dependence* and is denoted $S_1 \delta S_2$. There are two other data dependences which force us to keep execution order of two statements. In the program segment:

```
S1:a = b + c
S2:b = d / 2.
```

S_1 uses the value of b and S_2 reassigns a new value to b. Since S_1 uses an old value of b, S_1 should be executed before S_2. This is called *antidependence* and is denoted $S_1 \bar{\delta} S_2$. The third data dependence can be found in the following example:

```
S1:a = b + c
S2:d = a + 2.
S3:a = e + f
```

where S_1 assigns a value to a and S_3 reassigns a new value to a. Since the assignment of S_3 should be effective after the execution of this segment of the program, we can not change the execution order of S_1 and S_3. This is called *output dependence* and is denoted $S_1 \delta^{\circ} S_3$. In this example, we also have two other data dependences, $S_1 \delta S_2$ and $S_2 \bar{\delta} S_3$. The data dependence can be represented by a directed graph as shown in the following:



Since all three relations between S_1, S_2 and S_3 are fixed by the dependences, we can not change the execution order of these three statements nor we can not execute these in

parallel.

Analyzing data dependences, the flow of control should also be considered. For instance, in the following program:

```
S1:a = b + c
   if (x >= 0.)  then
S2:  d = a + 2.
   else
S3:  a = e + f
   endif
```

we have exactly the same statements as the previous example, S_1, S_2 and S_3 but with different flow of control. Although $S_1\delta^{\circ}S_3$ and $S_1\delta S_2$ hold in this example, $S_2\overline{\delta}S_3$ does not hold since we know either S_2 or S_3 is executed, but not both.

3.2 Distance and Direction Vectors

Inside loop structures, the data dependence relations become more complicated. Not only inter-statement dependences but also inter-iteration dependences can be exist since the same statement is executed many times. In the following example:

```
   for i = 1,n do
S1:  a(i) = b(i) + c(i)
S2:  d(i) = a(i) + d(i-1)
   endfor
```

S_2 depends on S_1 for all i, regardless of whether S_1 and S_2 are inside the loop. There is another dependence; S_2 depends on S_2, since $d(i-1)$ was computed by S_2 in the previous iteration. To distinguish the same statement in different iterations, we use superscripting. These dependences can be denoted $S_1^i\delta S_2^i$ and $S_2^{i-1}\delta S_2^i$. The *distance*

and the *direction* in the iteration space are defined on these relations. The iteration space *distance* between S_1^i and S_2^i is 0, and the *distance* between S_2^{i-1} and S_2^i is $i-(i-1)=1$. The *direction* is defined as follows:

$<$ The *distance* is always positive in the iteration space.

$=$ The *distance* is always zero in the iteration space.

$>$ The *distance* is always negative in the iteration space.

$*$ The *distance* can vary in the iteration space.

Using these *directions*, the relations in the example are usually denoted $S_1\delta_{=}S_2$ and $S_2\delta_{<}S_2$.

In multiple-nested loops, there is a *distance* for each loop. We make a tuple $(d_1,...,d_n)$ by putting all *distances* from outer-loop to inner. We call this tuple a *distance vector*. In a similar way, we can define the *direction vector* $(s_1,...,s_n)$. For instance, in the loop:

```
   for i = 1,n do
     for j = 1,n do
S1:    a(i,j) = a(i,j-1) + b(i,j)
S2:    c(i,j) = a(i,j) + d(i+1,j)
S3:    d(i,j) = 0.1
     endfor
   endfor
```

we can find the following dependences, *direction vectors* and *distance vectors*:

$S_1\ \delta_{(=,<)}\ S_1 \quad (=,<) \quad (0,1)$

$S_1\ \delta_{(=,=)}\ S_2 \quad (=,=) \quad (0,0)$

$S_2\ \overline{\delta}_{(<,=)}\ S_3 \quad (<,=) \quad (1,0)$

In this example, we have dependences with non-(=) directions in both columns of *direction vectors*. The outermost loop which has non-(=) direction dependences is usually called a dependence carrying loop, which has an important roll for loop restructuring discussed later in this chapter. Each loop in the example carries one dependence relation.

In the examples, we have constant *distances* so that *directions* can be found easily. In general, finding dependence and computing *distances* and *directions* are not trivial problems. We usually combine GCD test and Banerjee's Inequality[Ban76] to solve the common cases of these problems. So does Tiny[1].

3.3 Loop Restructuring Transformation

Using *direction vectors* and/or other results of the dependence analysis, Tiny can restructure loops without changing the semantics of the original program. Currently Tiny supports the following eight loop restructuring transformations[Wol90b]:

Parallelization Try to parallelize a loop. If there is no loop carrying dependence at the loop, this transformation succeeds.

Vectorization Try to vectorize a loop. If the loop is the innermost and it has no loop carried dependence cycle, this transformation succeeds.

Distribution Try to distribute a multi-statement loop. All dependence cycles should be kept in a single loop.

Interchange Try to interchange a loop with its immediate outer loop. If there is no $(<, >)$ dependence relation concerning the (outer, inner) loops, this succeeds.

Circulation Try to move the innermost(or outermost) loop to the outer-most (or innermost) position in a single step. A sufficient condition for this restructuring is discussed in [Wol91a].

[1]Tiny also has several more tests for more complex cases.

Skewing Add (or subtract) an outer loop index to the lower and the upper limits for an inner loop. This is always legal and can change its *direction vectors*.

Reversal Try to reverse the execution order of a loop. If the loop carries no dependence, this succeeds.

Bumping Add (or subtract) some constant integer to the upper and the lower limit of a loop. This is always legal.

More precise discussions and the algorithms for these loop restructuring are in [Wol91b]. More wide range of program restructuring based on the data dependence analysis are introduced in [PW86].

Chapter 4

Compilation

Our goal is to compile Tiny programs for Distributed Memory Multicomputers(DMMC). One obvious way to achieve the goal is to apply the methods used in the Crystal compiler to compile Tiny programs. This sounds straightforward, however there is a gap between the declarative (functional) language Crystal and the imperative language Tiny, which needs to be resolved somehow. In this chapter, we translate the Crystal compilation steps for an imperative language and apply them to Tiny programs. We use the same matrix multiply (MM) program which we used to illustrate the Crystal compiler. We also discuss the differences in each step between Crystal and Tiny.

4.1 Preprocess

Index Domain Alignment is a necessary preprocess step for Control Structure Synthesis in the Crystal compiler. Since a Crystal program only defines each *data field*, the compiler needs to relate the dimensions of all *data fields* to produce proper loop structures later. For imperative languages, this process is done by the programmer. In Figure 4.1, we show a Tiny version of the MM program. All necessary control structures including loops are already encoded in a program as you see in the figure. Although the Tiny compiler does not need this preprocess step, it may still be important to produce an efficient code. A program which is poorly written or hides the potential parallelism may be compiled to a slower DMMC program.

```
   real a(1:100,1:100)
   real b(1:100,1:100)
   real c(1:100,1:100)
   integer n

   for i = 1,n do
     for j = 1,n do
S1:    c(i,j) = 0.0
       for k = 1,n do
S2:      c(i,j) = c(i,j) + a(i,k) * b(k,j)
       endfor
     endfor
   endfor
```

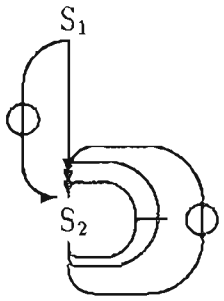
Figure 4.1: An MM Tiny program

4.2 Loop Restructuring

The second compiling step for the Crystal compiler is Control Structure Synthesis. The compiler derives loops and other constructs such that (1)the data dependence relations in programs are preserved and (2)the output fits in the data parallel model. Since a Tiny program already has a flow of control, we can translate this compiling step to restructuring a program such that (1) and (2) are satisfied. To satisfy (1), we need to analyze the data dependence relations of a program. We also need some systematic way to restructure loops for (2).

4.2.1 Data Dependence Analysis

After the Preprocess step, all Crystal program *data fields* are defined on the common *index domain* which has the largest dimensionality in the program. All functions are redefined on this common *index domain* and all loops are derived to be able to scan this. The Call-Dependence Graph (CDG) is used (1)to find functions which should be involved in the same loop (*strongly connected component*), (2)to decide the calculation order of each function (topological sort) and (3)to select the correct loop structures (breaking cycles). This graph is crucial to convert declarative descriptions to imperative codes.



$S_1\delta_{(=,=)}S_2$
$S_1\delta^{\circ}_{(=,=)}S_2$
$S_2\delta_{(=,=,<)}S_2$
$S_2\overline{\delta}_{(=,=,<)}S_2$
$S_2\delta^{\circ}_{(=,=,<)}S_2$

Figure 4.2: Data Dependence of MM program

For Tiny, a CDG is not necessary in the sense that the execution order and the loops are given as a program. We need the *data dependence* analysis to preserve the *data dependence* relations at this restructuring phase. We show the *data dependence* relations for the MM program in Figure 4.2. We use the built-in *data dependence* analysis of Tiny to find these relations. Note that the CDG and the *data dependence* graph (DDG) are quite different. The CDG shows the dependence relations of functions; by contrast the DDG shows the dependence relations between statements. Since the statements can change the values in memory more than once, the DDG has two more dependence types, *anti-dependence* and *output dependence*.

4.2.2 Program Decomposition

Following the Crystal compiler, first we want to find the strongly connected components in the DDG. Although we already have loops in Tiny programs, the structures and the kinds of loops may not be the same as those the Crystal compiler produces. To make these loop structures be similar to Crystal, we distribute all strongly connected components in the DDG into independent loops. In Figure 4.2, we have two strongly connected components, $\{S_1\}$ and $\{S_2\}$. Using the **Loop Distribution** transformation of Tiny, the MM program is converted as shown in Figure 4.3.

Even using the same strategy to form loops, the granularity of program decomposition is not exactly the same between Crystal and Tiny because of the differences between CDG and DDG. A DDG consists of statement nodes, whereas a CDG consists of definition

```
   real a(1:100,1:100)
   real b(1:100,1:100)
   real c(1:100,1:100)
   integer n

   for i = 1,n do
     for j = 1,n do
S1:    c(i,j) = 0.0
     endfor
   endfor
   for i = 1,n do
     for j = 1,n do
       for k = 1,n do
S2:      c(i,j) = c(i,j) + a(i,k) * b(k,j)
       endfor
     endfor
   endfor
```

Figure 4.3: A decomposed MM Tiny program

nodes. The definition of a function usually uses more than one statement so Tiny uses finer granularity at this decomposition. For instance in Figure 4.3, the two definitions of *c(i,j)* in S_1 and S_2 are divided into two loops. Finer granularity of Tiny does not mean smaller loops. Since the DDG has two more dependence relations which arise from the side effects of imperative languages, strongly connected components can be larger.

4.2.3 Loop Parallelization

To form a data parallel model, the Crystal compiler assumes that the common *index domain* is distributed among processors. Since all *data fields* are already aligned to the common domain and can be scanned from any dimension under the restriction of the CDG, the Crystal compiler tries to find the outermost dependence carrying loops which free all inner loops to be executed in parallel. As the result the outer dependence carrying domains become temporal domains and the rest become spatial domains which correspond to parallel loops.

Similarly, we assume that there is a virtual common array distributed among processors. We use the shape of an array which has largest dimensionality in the program as this virtual common array. We have not aligned all arrays to this common array yet, however we assume that we will be able to do it in the later restructuring phase. We want to form the loop structure such that the outermost loop carries dependences and the inner loops can be parallelized. We use the following algorithm to do this.

1. If all loops are processed then we are done.
2. Pick an unprocessed innermost loop.
3. Try to parallelize the loop by the **Loop Parallelization** transformation.
4. If the parallelization succeeds then repeat from 1.
5. If the parallelization fails then try the **Loop Interchange** transformation.
6. If the interchange succeeds then try to parallelize the loop.
7. Repeat from 1.

Since we use **Loop Parallelization** and **Loop Interchange** of Tiny, the *data dependences* in the original program are checked and preserved. In Figure 4.4, we show an output MM program of this phase. In the figure, the two loops around S_1 are parallelized. The k loop, a dependence carrying loop, is interchanged twice to become the outermost loop of S_2; consequently the two inner loops are parallelized.

4.3 Array Alignment

In the previous step, the Crystal compiler converted a program to intermediate imperative form with loop structures. The common *index domain* of the Crystal program is now divided into the *temporal domain* which is scanned by dependence carrying loops, and the *spatial domain* which forms parallel loop constructs. The Spatial Domain Alignment is the realignment of this common domain in the context of the given loop structure. Tiny

```
   real a(1:100,1:100)
   real b(1:100,1:100)
   real c(1:100,1:100)
   integer n

   doall i = 1,n do
     doall j = 1,n do
S1:    c(i,j) = 0.0
     endfor
   endfor
   for k = 1,n do
     doall i = 1,n do
       doall j = 1,n do
S2:      c(i,j) = c(i,j) + a(i,k) * b(k,j)
       endfor
     endfor
   endfor
```

Figure 4.4: A Parallelized MM Tiny program

programs have loop structures from the beginning. Since we assumed that the *common array* is distributed among processors and we restructured the loops in a similar way to Crystal, we want to align all arrays in the program to the *common array*. The objective is to find the array alignment which causes least communications during the program execution. To find the optimal alignment, we use the same strategy as Crystal, making a Component Affinity Graph (CAG) and finding its optimal partition.

4.3.1 Component Affinity Graph

The CAG is a weighted undirected graph built by the *reference patterns* of a Crystal program. Nodes of a CAG represent the *domain components* of Crystal *data fields*. From the inter-*data field* relations of the *reference patterns*, the *affinity relations* are derived and the weighted edges are drawn from this information. To built the CAG, we first need to derive *reference patterns* from a Tiny program. The Crystal compiler would already have found the *reference patterns* in order to build the CDG; since Tiny uses a DDG instead, the *reference patterns* will not have been found yet. The Crystal *reference*

c(i,j)←c(i,j)
c(i,j)←a(i,k)
c(i,j)←b(k,j)

Figure 4.5: Reference Patterns of MM program

patterns can be interpreted for Tiny as follows:

1. Find an assignment statement which assigns a value to an array A and is located inside a loop.

2. If the RHS of the assignment also has an array B then there is a reference from array B to array A.

3. If the assignment is under the control of conditional statements then all the arrays in the condition expressions should be considered as if they are in the RHS of the assignment.

In Figure 4.5, we show the *reference patterns* of the MM program. The index expressions in the *reference patterns* are most likely to have at least one loop variable; otherwise it does not make sense that the original assignment exists inside loops. Using exactly the same method used for Crystal, we can derive CAGs. The only difference is that the nodes or domain components in a Tiny CAG are not *data fields* but are arrays.

4.3.2 Optimal Partition of CAG

Using exactly the same heuristic algorithm as Crystal, we can partition the CAGs. We show the partitioned CAG of the MM program in Figure 4.6. In the figure, we have three columns and two rows since there are three arrays, *a*, *b* and *c* in the *reference patterns* and the largest array is two dimensional. From the second and the third *reference patterns*, we derived two edges, *(dom(c,1), dom(a,1))* and *(dom(c,2), dom(b,2))*. All edges are rated as 1 since neither is along the *temporal component* nor a competing edge.

Using the *align* statement we added to Tiny, the aligned MM program is derived as shown in Figure 4.7. The Tiny compiler selects the largest and the latest declared array

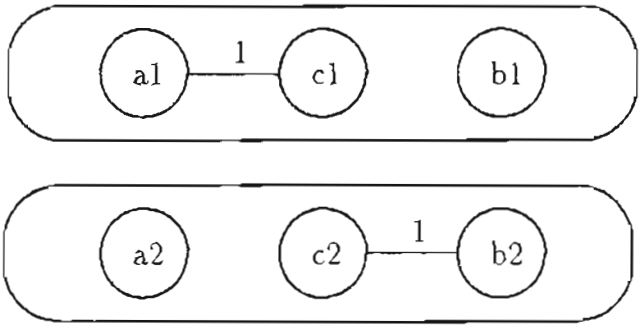


Figure 4.6: Partitioned CAG for MM program

```
   real a(1:100,1:100)
   real b(1:100,1:100)
   real c(1:100,1:100)
   integer n

   align(a(#1,#2),c(#1,#2))
   align(b(#1,#2),c(#1,#2))

   doall i = 1,n do
     doall j = 1,n do
S1:    c(i,j) = 0.0
     endfor
   endfor
   for k = 1,n do
     doall i = 1,n do
       doall j = 1,n do
S2:      c(i,j) = c(i,j) + a(i,k) * b(k,j)
       endfor
     endfor
   endfor
```

Figure 4.7: An Aligned MM Tiny program

as the virtual processor array. In the MM program, the array c is selected and used as a reference in the two *align* statements. Any Tiny program can be converted to this aligned form automatically.

Chapter 5

Issues of Program Restructuring

After implementing the Crystal compiling steps in Tiny, we tested the transformations by converting several Tiny programs to the parallelized and aligned forms. We have found several problems during the experimental tests. In this chapter, we discuss these problems and the possible solutions.

5.1 Array Alignment vs Spatial Domain Alignment

We find a problem in the MM program which we used to illustrate both the Crystal compilation steps and the Tiny compilation steps. The outputs of these compilations show the different alignments. Using the Tiny *align* statements, the alignments can be described as follows:

Crystal Output	Tiny Output
`align(a(#1,#3,1),c(#1,#2,#3))`	`align(a(#1,#2),c(#1,#2))`
`align(b(#3,#2,1),c(#1,#2,#3))`[1]	`align(b(#1,#2),c(#1,#2))`

Note that the *align* statements in the Crystal output include the third dimension #3 which are not distributed among processors but stored in each virtual processor. We defined the *align* statement as aligning an array to the common virtual processor array in the program. We selected the array which has the highest dimensionality as the

[1] The direct translation from Figure 2.9 would be align(b(#2,#3,1),c(#1,#2,#3)) though the b is transposed in the Crystal program.

common array. This common array is comparable to the *spatial domain* in a Crystal program. The Tiny array alignment forced all the arrays to be aligned inside the spatial domain. This is wrong since we sometimes want to keep values in the same processor along some dimension of an array. The most common such case is that the domain we want to keep values along is scanned by a sequential loop, which is comparable to the Crystal *temporal domain.* The Crystal compiler does not have this problem because even at the Spatial Domain Alignment phase, it aligns all *data fields* to the common *index domain* which has both the *spatial* and *temporal domains.*

We thought that we interpreted the Crystal Spatial Domain Alignment correctly as the Tiny Array Alignment, though we had incorrectly dropped the *temporal domain* part from a Tiny program. The next question is "Where can we find the temporal domain in a Tiny program?" The answer is "In the sequential loops." We can't find the temporal domain in the Tiny arrays as we could find in the Crystal *data fields* because we do not need it. For instance, in the following Crystal program:

```
dom T = [0..n]
dfield fact(i): T = if (i = 0) then 1.0
                  || else fact(i-1) * i
                  fi
```

we need the *index domain T* to express the data dependence relations in the definition of *fact.* This turns out to be the *temporal domain* because the domain carries the *call-dependence.* A naive Tiny interpretation of this program would be as follows:

```
real fact(0:100)
integer n
fact(0) = 1.0
for i = 1,n do
  fact(i) = fact(i-1) * i
endfor
```

But in a Tiny program we do not need the domain which corresponds to T in the Crystal program since we are not defining the function; instead we can just assign a value and overwrite the value. The more realistic equivalent program is as follows:

```
real fact
integer n
fact = 1.0
for i = 1,n do
  fact = fact * i
endfor
```

In this program, *fact* is defined as a scalar. Although we have no arrays in the program, *fact* is reassigned n times and therefore we can see a sequence of assignments to *fact* along a temporal domain. We missed this case because we only collected the information concerning the arrays in Tiny programs.

We can fix this problem using the loop iteration spaces in a Tiny program as domains for a variable. We can relabel a variable inside loops such that the variable has all the loop variables as indices. The relabeling algorithm is roughly as follows:

1. Find an assignment which is located inside loops.
2. Add all the loop variables as indices to a variable in the LHS of the assign statement as if it is an array that has exactly the same dimensionality as the nesting level of the statement.
3. Change the all variables for which new indices are added and is in the same level of loop nesting as its RHS counterpart. (If a new loop variable is added to LHS, the variable should be decremented by the step of the loop in RHS.)

We can add the relabeling step followed by the steps we already developed for Tiny programs. For instance relabeled S_2 in Figure 4.4 would be:

```
c(i,j)(k,i,j) = c(i,j)(k-1,i,j) + a(i,k) * b(k,j)
```

```
   real a(1:100,1:100)
   real b(1:100,1:100)
   real c(1:100,1:100)
   integer n

   align(a(#2,#1),#2(#1,#2,#3))
   align(b(#1,#3),#2(#1,#2,#3))
   align(c(#2,#3),#2(#1,#2,#3))

   doall i = 1,n do
     doall j = 1,n do
S1:    c(i,j) = 0.0
     endfor
   endfor
   for k = 1,n do
     doall i = 1,n do
       doall j = 1,n do
S2:      c(i,j) = c(i,j) + a(i,k) * b(k,j)
       endfor
     endfor
   endfor
```

Figure 5.1: An Aligned MM Tiny program

The second pseudo indices are added to the *c(i,j)* references. In the following steps, we use these pseudo indices to create *reference patterns* and to build the CAG. Finding the deepest nested loop structure and selecting it as a common *index domain* for the Tiny program we can derive the same aligned program as the Crystal program as shown in Figure 5.1. Each array is aligned to an iteration space domain, similar to a Crystal domain. A domain denoted $\#i(\#j_1,..,\#j_n)$ to refer to the i-th independent loop structure, and $\#j_1,..,\#j_n$ refers to the nested loops of that structure. In Figure 5.1, *#2(#1,#2,#3)* refers to the second loop structure (the *k-i-j* loop).

5.2 Optimal Program Structure

In the loop restructuring phase, we tried to find the outermost *dependence carrying* loop to parallelize as many inner loops as possible. The algorithm we used is not the optimal

because of the following two reasons:

1. We have not tried to reduce cycles which are made by *anti-dependence* and/or *output dependence*, which Crystal programs do not have. These cycles can be eliminated through additional memory usage and data copying.

2. We have not taken all possible loop transformations to get more parallelism. For instance, we could use the Tiny **Loop Skewing** transformation to parallelize loops which have *flow dependence* cycles in programs.

We have to describe two more things we have not considered. First we only use the *doall* construct for this project. Tiny has another construct *forall* which is comparable to the *FORALL* of Fortran-D. This may give us more appropriate representation of parallelism. Second we have not discussed the optimization of mapping parallel loops to physical processors. Our strategy was (1)Assume all array elements are distributed among all (virtual) processors, (2)Distribute virtual processors to physical processors, and (3)Simulate virtual processors by sequential loops in each node. We do not know how we can convert all parallel loops into efficient virtual processor simulation loops.

Further more, comparing the Tiny output of the MM program to one which is hand-coded for a DMMC[Ott91], we found more sophisticated data movement in the latter program. So far we assumed that arrays in a program will be statically distributed in blocked interleaved fashion after the alignment phase. However in that program, Otto dynamically redistributes the arrays such that each processor cycles through the entire array. Assuming that n is the number of processors, Otto adds a new sequential outermost loop which changes the distribution n times. A block of array data travels around all processors in n iterations. Although this schema does not break our alignment policy, we don't have the method to form this kind of loop structure.

Chapter 6

Implementation

The compilation steps we have described are implemented as one of the menus of the Tiny loop restructuring tool. An user can select Crystal from the menu, and can perform all compilation steps at once or separately step by step. The compilation steps are divided into four programs, (1)Program Decomposition, (2)Loop Parallelization, (3)CAG Building, and (3)CAG Optimal Partioning. In this chapter we show these four algorithms.

6.1 Program Decomposition

The Tiny compiler parses an input program and creates a symbol table and an abstract syntax tree. Each node of the syntax tree has two special pointers for a list of incoming dependence relations and a list of outgoing dependence relations. An element of the relation list consists of a pointer to a node which is related by the dependence and a direction vector for the dependence. The Tiny compiler can handle its loop restructuring transformations using this data dependence information. We use these loop restructuring subroutines for the compilation for DMMC. In Figure 6.1, we show an algorithm for the program decomposition phase. The procedure *Traverse_Distribute* simply traverses the abstract syntax tree in the depth first fashion and tries to distribute loops using the Tiny builtin procedure *Distribute_Loop*.

Algorithm Decompose(*Entry*)
Input: Entry node *Entry* of a Tiny abstract syntax tree.
Output: A decomposed Tiny abstract syntax tree.
begin
  Traverse_Distribute(*Entry*)
end

Procedure Traverse_Distribute(*N*)
Input: A node *N* of a Tiny abstract syntax tree.
Output: A decomposed Tiny abstract syntax tree.
begin
  if a node N is null **then return**
  Traverse_Distribute(*N*→*Child*)
  Traverse_Distribute(*N*→*Next*)
  if a node N is a loop **then**
    Distribute_Loop(*N*)
end

Procedure Distribute_Loop(*N*)
Input: A loop node *N* of a Tiny abstract syntax tree.
Comment: Using data dependence information, try to distribute the nodes belong to the loop.

Figure 6.1: Program Decomposition Algorithm

6.2 Loop Parallelization

In Figure 6.2, we show an algorithm for the loop parallelization phase. The procedure *Traverse_Parallelize* also traverses the abstract syntax tree in depth first fashion and tries to apply *Parallelize_Loop* and/or *Interchange_Loop* for each loop. Because of the side effect of the loop interchange, the algorithm becomes slightly complicated. When the loop interchange is valid, we have to traverse nodes next to the new inner loop since we will not visit the node again as we would do if no interchange occurred. Although not shown in Figure 6.2, *Traverse_Parallelize* counts valid loop interchanges. In this algorithm, we continue to call *Traverse_Parallelize* while the count is not zero. This is somewhat similar to the bubble sort. Again because the Tiny functions *Parallelize_Loop* and *Interchange_Loop* take care of all data dependence constraints, the algorithm was implemented easily.

6.3 CAG Building

To find a good alignment of arrays, we build a CAG from reference patterns in a program. We added two data structures for components and edges to Tiny. The CAG nodes are represented by multiple columns of the components. Each component has pointers to incoming and outgoing edges for affinity relations. Each edge has a weight. In the Tiny symbol table, we added a pointer to the column of components for each array entry. In Figure 6.3, we show an algorithm to build the CAG. *Make_Component* prepares all data structures needed in the following procedures. *Traverse_Build_CAG* traverses the abstract syntax tree and finds reference patterns. *Fix_Edgeweight* checks all edges and changes the weights if the edge connects two temporal components.

In Figure 6.4, we show *Traverse_Build_CAG*. *Traverse_Build_CAG* tries to find two particular statements, *assign* statement and *if* statement. From an *assign* statement, it searches reference patterns by scanning RHS expressions and all conditional expressions which have a control dependence to the assignment. From a *if* statement, it pushes a pointer to conditional expressions.

```
Algorithm Parallelize(Entry)
Input: Entry node Entry of a Tiny abstract syntax tree.
Output: A loop restructured Tiny abstract syntax tree.
begin
   repeat
      Traverse_Parallelize(Entry)
   until no interchanges
end

Procedure Traverse_Parallelize(N)
Input: A node N of a Tiny abstract syntax tree.
Output: A loop restructured Tiny abstract syntax tree.
begin
   if a node N is null then return
   Traverse_Parallelize(N→Child)
   Traverse_Parallelize(N→Next)
   if a node N is not a loop then return
   if Parallelize(N) fails then begin
      find an immediate outer loop N2
      if Interchange_Loop(N) succeeds then
         if Parallelize_Loop(N2) fails then
            /* restore the loop structure */
            Interchange_Loop(N2)
         else
            Traverse_Parallelize(N→Next)
   end
end

Procedure Parallelize_Loop(N)
Input: A loop node N of a Tiny abstract syntax tree.
Comment: Using data dependence information, try to parallelize the node. If parallelization fails, return false.

Procedure Interchange_Loop(N)
Input: A loop node N of a Tiny abstract syntax tree.
Comment: Using data dependence information, try to interchange the loop node with its immediate outer one. If interchange fails, return false.
```

Figure 6.2: Loop Parallelization Algorithm

```
Algorithm Build_CAG(Entry)
Input: Entry node Entry of a Tiny abstract syntax tree.
Output: A CAG of the Tiny program.
begin
   Make_Component()
   Traverse_Build_CAG(Entry)
   Fix_Edgeweight()
end

Procedure Make_Component()
Output: Components for all dimensions of all arrays and a pointer to the common array
CT and its dimensionality MaxT.
begin
   for all variables in the symbol table
      if the variable is an array then begin
         make component nodes for all dimensions of the array
         if the dimensionality is greater or equal to MaxT then
            set MaxT to the dimensionality and set CT to the component node pointer
      end
end

Procedure Fix_Edgeweight()
Input: A CAG with edges weighted ε and 1.
Output: A complete CAG with all kinds of edges.
begin
   for each component in the CAG
      if the component is temporal then
         for each affinity edge
            if the counterpart component is sequential then
               change the edge weight to ∞
end
```

Figure 6.3: CAG Building Algorithm

```
Procedure Traverse_Build_CAG(N)
Input: A node N of a Tiny abstract syntax tree.
Output: Affinity edges between components which form a CAG.
begin
   if a node N is null then return
   if a node N is an assignment statement then begin
      RHS = N→Child
      LHS = N→Child→Next
      if a node LHS is an array then
         for each dimension of the array
            if an index expression has a loop variable then begin
               /* try to find reference patterns */
               traverse all RHS nodes to make the affinity edges
               /* we have to see all condition expressions */
               for I = 0 to IF_Nest
                  traverse Cond[I] nodes to make the affinity edges
            end
      Traverse_Build_CAG(N→Next)
      return
   end
   if a node N is an if statement then begin
      Cond[IF_Nest ++ ] = N→Child
      Traverse_Build_CAG(N→Child)
      IF_Nest --
      Traverse_Build_CAG(N→Next)
      return
   end
   Traverse_Build_CAG(N→Child)
   Traverse_Build_CAG(N→Next)
end
```

Figure 6.4: CAG Building Algorithm

6.4 CAG Partitioning

We use exactly the same heuristic algorithm as the Crystal compiler to partition a CAG. In Figure 6.5, we show this algorithm. We iterate a sequence of three procedures, *Form_Bipartite_Graph*, *Optimal_Alignment* and *Reduce_Graph*, until all columns of a CAG are processed. *Form_Bipartite_Graph* makes a bipartite graph GX from a CAG. The optimal partition of GX, which is the maximum bipartite matching M[CLR90], is calculated by *Optimal_Alignment*. *Reduce_Graph* combines CT and CX to reduce the CAG G. At every iteration an align statement is created from M and is inserted to the original Tiny program.

```
Algorithm Partition_CAG(G)
Input: A CAG G and a column of CAG which represent the common array CT.
Output: An aligned Tiny program.
begin
   get pointers to all columns of the CAG and make a list Clist
   Unlink CT from Clist
   for each column CX in Clist begin
      GX = Form_Bipartite_Graph(CT,CX,G)
      M = Optimal_Alignment(GX)
      G = Reduce_Graph(CT,CX,G,M)
      Insert an align statement specified by the mapping M
      Unlink CX from Clist
   end
end

Procedure Form_Bipartite_Graph(CT,CX,G)
Input: A CAG G, a pointer to the common array column CT and a pointer to an array
column CX.
Output: Bipartite graph GX between CT and CX.
begin
   GX = empty
   add all nodes from CT and CX to GX
   for each dimension node DX in CX
      for each dimension node DT in CT
         if there is a path between DX and DT then begin
            add an edge E between DX and DT in GX
            sum all edge weights in the connected components of DX in G
            set the weight of E to this sum
         end
end

Procedure Optimal_Alignment(GX)
Comment: This is an algorithm to find maximum bipartite matching[CLR90].

Procedure Reduce_Graph(CT,CX,G,M)
Comment: Using alignment M, combine all nodes in CX to those in CT. Two edges
remaining between the same nodes should be replaced by a single edge weighted by the
sum of the weights of those edges and all self cycles should be eliminated.
```

Figure 6.5: CAG Partitioning Algorithm

Chapter 7

Conclusion

Automatic program restructuring for Distributed Memory Multicomputers is discussed and is implemented to the Tiny loop restructuring tool as one of its menu selection.

We interpret the compilation steps for a functional language Crystal to an imperative language Tiny and implement the steps except for the last communication generation phase to Tiny. During this process we found:

1. Crystal Index Domain Alignment step is a necessary preprocess for Control Structure Synthesis. Since any Tiny program has control structure, we do not need this step to compile Tiny programs.

2. The Crystal compiler creates loop structures in Control Structure Synthesis step. To restructure a Tiny program to have similar loop structures to Crystal we need to use **loop distribution** transformation, **loop interchange** transformation and **loop parallelize** transformation.

3. Crystal Spatial Domain Alignment is an effective step to get efficient interprocessor communication. Traversing loop structures to find array assignments and making the *reference patterns* in a Tiny program, we can build CAGs to use the same method as Crystal for array alignment of Tiny programs.

Converting several Tiny programs by the method we developed, we found the following issues:

1. Array alignment we used for Tiny tries to align all arrays to a common array, which is the alignment only inside *spatial domain*. To get the same result as Crystal, we need to align arrays with *temporal domain*, which is a loop in Tiny.

2. The loop restructuring we did is not optimal. We have other loop transformations which we did not considered.

The Crystal compiler creates loop structures from *reference patterns* of Crystal declarative definitions of functions therefore the loop structure inherently follows data parallel model. Since the loop structure of a Tiny program depends on how a programmer writes it, the loop restructuring has an important role for the compilation. Although we could derive the loop structure similar to Crystal, it is not clear that either the structure is the best for Tiny or not.

Bibliography

[Ban76] Utpal Banerjee. Data dependence in ordinary programs. M.S. thesis UIUCDCS-R-76-837, Univ. Illinois, Dept. Computer Science, November 1976.

[CK87] David Callahan and Ken Kennedy. Analysis of interprocedure side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550, October 1987.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, chapter 27.3, page 601. The MIT Press, 1990.

[FHK+90] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran d language specification. Technical Report TR90-141, Rice Univ., December 1990. Revised April,1991.

[HS86] W. D. Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[ISO91] ISO. *Fortran 90*, May 1991.

[Kar87] Alan H. Karp. Programming for parallelism. *IEEE Computer*, 20(5):43–57, May 1987.

[Li91] Jingke Li. *Compiling Crystal for Distributed-Memory Machines.* PhD dissertation, Yale University, Department of Computer Science, October 1991.

[OCA86] R R. Oldehoeft, D C. Cann, and S J. Allan. Sisal: initial mimd performances results. In Wolfgang Handler, Dieter Haupt, Rolf Jeltsch, Wilfried Juling, and Otto Lange, editors, *CONPAR 86Proc. of the conference on algorithms and hardware for parallel processing*, number 237 in Lecture notes in computer science, pages 120–127, Aachen, Germany, September 1986. Springer-Verlag New York, Inc., New York, NY.

[Ott91] Steve W. Otto. Metamp: A higher level abstraction for message-passing programming. Unpublished, January 1991.

[Par91] Parallel Computing Forum. *PCF Parallel Fortran Extensions*, July 1991.

[PW86] David A. Padua and Michael Wolfe. Advanced compiler optimizations for supercomputers. *Communications ACM*, 29(12):1184–1201, December 1986.

[Wol78] Michael Wolfe. Techniques for improving the inherent parallelism in programs. M.S. thesis UIUCDCS-R-78-929, Univ. Illinois, Dept. Computer Science, July 1978.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. Pitman Publishing, London, 1989. (also available from MIT Press).

[Wol90a] Michael Wolfe. A loop restructuring research tool. Technical Report CSE 90-014, Oregon Graduate Institute, August 1990.

[Wol90b] Michael Wolfe. *TINY A Loop Restructuring Research Tool.* OGI, December 1990.

[Wol91a] Michael Wolfe. Experiences with data dependence abstractions. In *Proc. 1991 International Conf. on Supercomputing*, pages 321–329, Cologne, June 1991.

[Wol91b] Michael Wolfe. The Tiny loop restructuring research tool. In *Proc. 1991 International Conf. on Parallel Processing*, volume II, pages 46–53, St. Charles, IL, August 1991. Penn State Press.

Biographical Note

Mitsuru Ikei was born on January 5, 1957 in Kagoshima City which is located in the southernmost prefecture in Kyushu Island in Japan.

He entered Kagoshima La Salle High School in 1969 and graduated in 1975. After graduation, he moved to Yokohama City to enter Yokohama National University, where he got his B.S. in Electrical Enginnering in 1982. He joined Hitachi Chemical Company Ltd., in 1982 and since then he has been working as a researcher at the Shimodate Research Laboratory in Shimodate City.

In 1990, he came to the US to work with the engineers of Cogent Research, an Oregon based parallel computer company, to parallelize computational chemistry programs. He enrolled in the graduate program in Computer Science & Engineering at OGI in September 1990.