Automatic Incorporation of Modifications to Sequential Code in Efficient Compile-Time Parallelization on Distributed-Memory Machines

A thesis
submitted in partial fulfillment
of the requirements for the degree
of

Bachelor of Technology
in
Computer Science and Engineering

by

Pankaj Gupta

Neeraj Mittal

Under the guidance of

Dr. Pankaj Mehra
Dr. Gautam M. Shroff

Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

May 1995

Certificate

This is to certify that the thesis titled *Automatic Incorporation of Modifications to Sequential Code in Efficient Compile-Time Parallelization on Distributed-Memory Machines* being submitted by *Neeraj Mittal* and *Pankaj Gupta* to the Indian Institute of Technology, Delhi, for award of the **Bachelor of Technology** in **Computer Science and Engineering** is a record of bonafide work carried out by them under our supervision.

The results presented in this thesis have not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Pankaj Mehra
Visiting Professor
Department of Computer Science
Indian Institute of Technology
New Delhi 110016

Gautam M. Shroff
Assistant Professor
Department of Computer Science
Indian Institute of Technology
New Delhi 110016

To Our Parents

Abstract

MIMD distributed-memory machines are capable of providing enormous computational power. However, the difficulty of developing parallel programs for these machines has limited their use. The most difficult part is essentially of determining a suitable data distribution scheme for a program. Most of the current projects and parallel tools leave this problem entirely to the user.

Our thesis is that even though completely automatic parallelization of a sequential program may be extremely difficult, the process of incorporating simple modifications to sequential code, given a formally documented parallel code for the original sequential program, can be *fully* automated. To validate this thesis, we have implemented an advanced tool that can efficiently parallelize such a modified sequential code, given a formal description of the data-decomposition scheme employed in parallelizing the original sequential code.

Our system is organized around three major functions: *program analysis and restructuring*, *data-decomposition determination* and *compile-time code generation*. Program analysis and restructuring is done with the aid of *SIGMA II*, a toolkit for building parallelizing compilers and performance-analysis systems. Determination of data decompositions is achieved by alignment with those variables whose decompositions are already known (from the given parallelization of the old sequential code). A novel strategy has been used for the purpose of alignment. Efficient compile-time code generation is achieved by means of an advanced compiler that performs compile-time resolution and interprocedural compilation. Some optimizations are also made to the generated parallel program.

Acknowledgements

We would like to convey our deeply felt gratitude towards Dr. Pankaj Mehra and Dr. Gautam. M. Shroff, for providing us the opportunity to work on this project. They were a constant source of inspiration and encouragement, and their superlative guidance and impeccable technical advice was of immense help in the culmination of this work. They showed us light on every dark corner, and we duly appreciate their patience and motivating drive.

We also extend thanks to Prof. S.N. Maheshwari. His pertinent insights were of tremendous help in fine tuning our efforts.

Our thanks to all those seniors who contributed in building up the base of knowledge on top of which we performed this work. Gaurav Banga, Gagan Hasteer, Asheesh Sharma and Saurab Nog and all others made valuable contributions. This work owes much to the cooperation extended to us by *Weather Team:* Sanjay Agarwal and Arun Chauhan. Our many thanks to Mohit Aron and Ashok Bhatia. They were a constant source of encouragement and proved to be great friends. We will forever cherish their friendship.

Last but not the least, we appreciate the cooperation we received from Mr. Jaswant and Mr. Prasad regarding laboratory activities.

Neeraj Mittal Pankaj Gupta

Contents

1 Introduction **2**

1.1 Background 2

1.1.1 Machine-independent parallel programming 2

1.1.2 Compiler assistance 3

1.2 Motivation 3

1.3 Thesis 4

1.4 Overview of the System 5

1.5 Organization of the thesis 8

2 Fortran D Language **9**

2.1 The Data-Parallel Programming Model 9

2.2 DIMENSION Statement 10

2.3 DECOMPOSITION Statement 10

2.4 LINEARIZED Statement 11

2.5 ALIGN Statement 11

2.5.1 Exact match 13

2.5.2 Intra-dimension alignment 13

2.5.3 Inter-dimension alignment 14

2.6 DISTRIBUTE Statement 15

2.7 Discussion 17

3 SIGMA II **18**

3.1 Introduction 18

3.2 The SIGMA Architecture 19

3.2.1 The database structures 20

3.2.2 Low-level expressions 22

3.3 Discussion 22

4 User Annotations **23**

4.1 DISTRIBUTION FILE IS Statement 23

4.2 IDENTICAL Statement 24

4.3 SIMILAR Statement 24

4.4 LINEARIZED Statement 24

4.4.1 Delinearization 25

4.5 TREAT AS NORMAL Statement 28

4.6 RECURRENCE Statement 28

4.6.1 Compile-time recursion 28

4.6.2 Run-time recursion 29

4.6.3 An extended example for the linearized and recurrence statements 29

4.7 REDUCE Statement 30

4.8 Discussion 30

5 Determination of Data Distributions **32**

5.1 Data Distribution 32

5.2 Alignment and Detection of Communication Requirements 35

5.2.1 Alignment 36
5.2.2 Detection of communication requirements 38
5.2.3 Solving the detection of communication problem precisely . . 39
5.3 Overall strategy of distribution determination 42

6 Theoretical Issues in Code Generation 43
6.1 Compilation Example 43
6.1.1 Run-time resolution 44
6.1.2 Compile-time resolution 44
6.2 Formal Model 44
6.2.1 Distribution functions 45
6.2.2 Regular distributions 45
6.2.3 Computation 45
6.2.4 Image, local index sets 46
6.2.5 Iteration sets 46
6.3 Parallel Code Generation 46
6.3.1 Run-time resolution 47
6.3.2 Compile-time resolution 48
6.4 Interprocedural Compilation 50
6.4.1 Reaching decompositions 50
6.4.2 Array reshaping 52
6.5 Discussion 52

7 Conclusion and Directions for Future Work 53
7.1 Summary and Current Status 53
7.2 Future Work 54

7.2.1 Removal of user annotations 54
7.2.2 Removal of communication free restraint 54
7.2.3 Optimizations 54
7.2.4 Automatic data partitioning 55
7.2.5 Automatic parallelization 55

A Tutorial Introduction to Our System **56**

List of Figures

1.1 System Objective 6

1.2 System Organization 7

2.1 Linearized Arrays 12

2.2 Some Fortran D Constructs 16

4.1 A Linearized Array composed of several L-sections 25

4.2 An L-section of a Linearized Array 26

4.3 The $(c+1)^{st}$ Strip 27

5.1 Different data partitions for a 16 * 16 array 34

A.1 Sample view of the user interface 57

Chapter 1

Introduction

Distributed memory multiprocessors (multicomputers) are increasingly being used for providing high levels of performance for scientific and engineering applications. Distributed memory machines offer significant advantages over their shared memory counterparts in terms of cost and scalability, but their power of massively-parallel processing is hindered by the difficulty of parallel programming. Because of the absence of a single global address space, the programmer has to distribute code and data on processors himself. Automation of this process is therefore both desirable and necessary for widespread use of these machines to become a reality.

1.1 Background

1.1.1 Machine-independent parallel programming

It is widely recognized that parallel computing represents the only plausible way to continue to increase the computational power available to scientists and engineers. Highly parallel supercomputers also provide the best cost/performance ratio of all supercomputers. Despite their advantages, parallel machines have only enjoyed limited success because parallel programming is a difficult and time-consuming task. To obtain adequate performance, scientific programmers must write explicitly parallel programs and solve many machine-dependent issues.

Because of the difficulty of writing parallel programs and making modifications to

it scientists are generally discouraged from utilizing parallel machines.

1.1.2 Compiler assistance

The goal of this thesis is to solve the parallel programming problem by developing the technology needed to make the process of making modifications to an existing parallel code easy. More precisely the objective is to automate the parallelization of new sequential code which is a *modified* version of old sequential code. It is assumed that the old sequential code has been parallelized manually since we feel that the efficiency of the parallel program generated by an automatic parallelizing tool cannot equal that obtained by hand parallelization (This statement is validated by the non-existence of fully automatic parallelization tools.) Moreover hand parallelization of the skeleton sequential code is a small price to pay for gaining facility to making any modifications to this code easy. This process also fits naturally into the practical model of program development, wherein a programmer likes to incrementally build his program.

Our approach would be to identify a *data-parallel* programming style for Fortran that may be compiled to execute efficiently. However in converting from a Fortran program, the compiler simply is not able to always do a good job of picking the best alternative in every tradeoff, particularly since it must work solely with the text of the program. As a result, the programmer may need to add additional information in the form of *user annotations* to the program for it to be correctly and efficiently parallelized. The programmer expresses the parallelism in terms of annotations and data distribution functions for the various arrays of the program. Our tool then uses these distribution functions to generate parallel code for this program.

1.2 Motivation

The goal of automatic parallelization of sequential code remains incomplete as long as the programmer is forced to think about the issues such as coming up with the right data partitioning scheme for each program. The task of determining a good partitioning scheme manually can be extremely difficult and tedious. However, most of the existing projects on parallelization systems for multicomputers have so far chosen not to tackle this problem at the compiler level (or tackle it in a very restricted

sense) because it is known to be an extremely difficult problem. Another related problem, the component alignment problem has been discussed by [12], and shown to be NP-complete. On the other extreme, recently several researchers [11] have addressed the problem of automatically determining a data partitioning scheme, but have largely restricted themselves to a small class of programs, which often does not include scientific applications. Hence completely automatic determination of data partitioning schemes has not been satisfactorily achieved till date. In this scenario it becomes imperative to take an in-between course and automate the process of making modifications to sequential code for the purpose of efficient compile-time parallelization of the modified sequential code. This is the precisely the objective of this thesis.

The immediate motivation for this project is derived from the ongoing Weather Project in the department which involves parallelization of the T-80 Cray Fortran code for the PARAM. The sheer size of this code –about 32,000 lines of Fortran– coupled with its complexity makes the use of an automatic tool both desirable and imperative.

1.3 Thesis

We believe that a fundamental ingredient required for compiling programs for distributed-memory machines is a specification of the data decomposition of the program. In others words, we assume that when data decompositions or some additional information from which these can be inferred are provided for sequential programs, an advanced compiler can generate parallel programs that execute efficiently on MIMD distributed-memory machines. When data distributions of all the variables are not known, we can still generate efficient parallel programs by doing program analysis and finding out the relation of these variables with those whose distributions are known. Our thesis is that even though completely automatic parallelization of a sequential program may be extremely difficult, the process of making modifications to sequential code for generating parallel programs can be *fully* automated. To validate this thesis, we have implemented an advanced tool that can efficiently parallelize such a modified sequential code, given a formal description of the data-decomposition scheme employed in parallelizing the original sequential code.

1.4 Overview of the System

The basic aim is to automate the parallelization of sequential code using the SPMD programming paradigm. Since completely automatic parallelization is still not possible, our system uses previously parallelized codes to aid itself in the process of parallelization. Given a sequential code and its parallelized version (assumed to be parallelized manually or interactively), the system (see figure 1.1) will automatically parallelize (*efficiently* i.e. at compile-time) any new sequential code (that has some *relationship* with the original code). This *relationship* can be in terms of some modifications in the original sequential code like addition (deletion) of new (old) variables, different access patterns, addition (modification) of new (old) subroutines etc. The system uses information from the *information* file which has description of distributions of array variables in the original code.

The system is developed for the sequential code of the ongoing weather project and hence some specific issues merit attention :

1. The resulting parallel code is desired to be **communication free**. This restriction is introduced to make the alignment procedure simpler. If this is not assumed then the techniques mentioned in this thesis will still apply, though with some modifications. The assumption of no communications merely adds to our convenience in generating more efficient parallel programs than would have been otherwise possible, although the decision to introduce this restriction arises partly from the belief that handling of communications is best (i.e. most efficiently) achieved when left to the programmer, since an automatic tool would not be able to perform as many communication optimizations as might be required/desired for the particular scientific or engineering application.

2. **Linearized arrays** (arrays whose dimensions have been collapsed for efficient storage utilization) require specific treatment.

3. It is assumed that some **user annotations** are present in the code which aid the system in making decisions regarding data distributions and code generation.

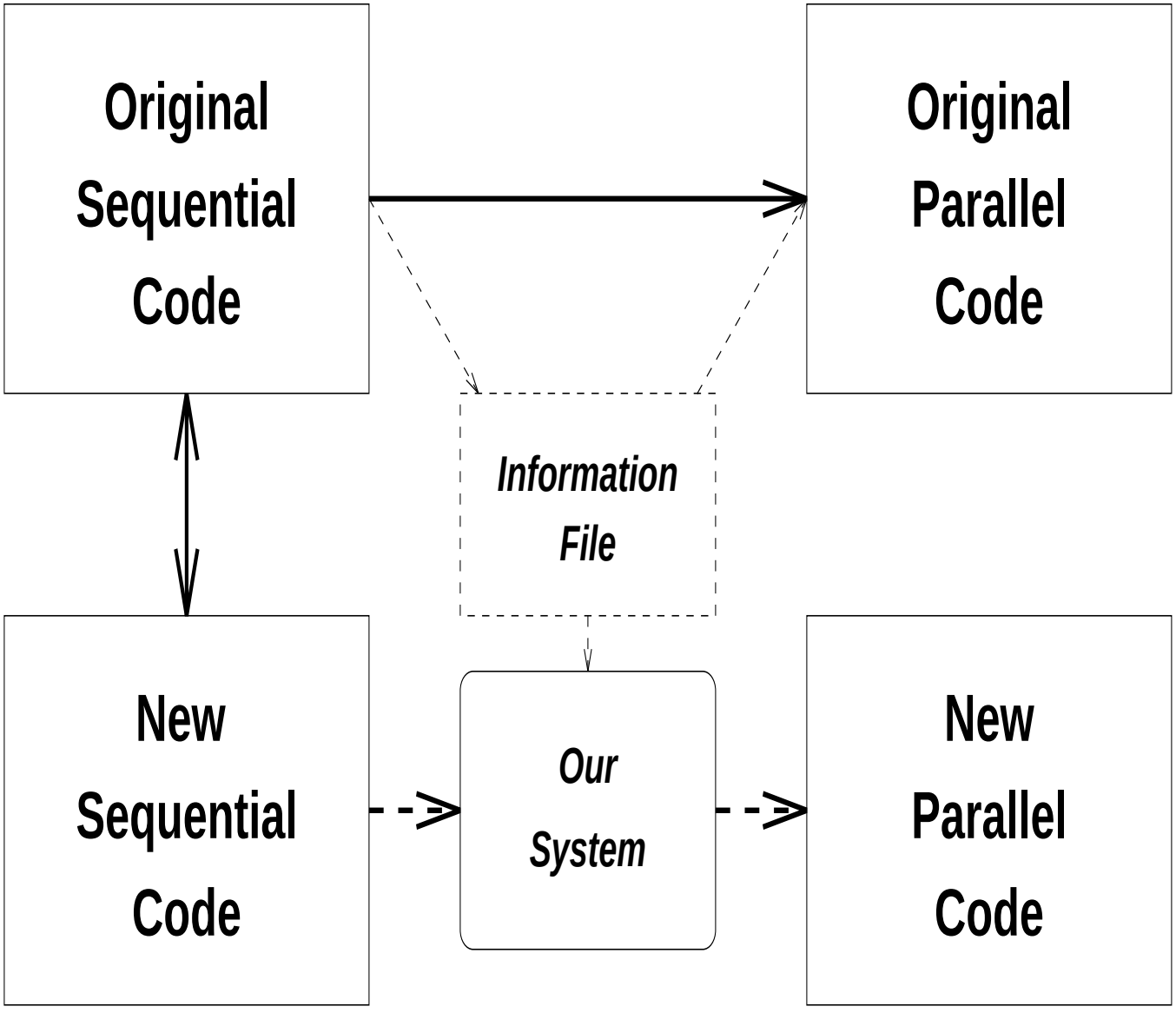

Figure 1.1: System Objective

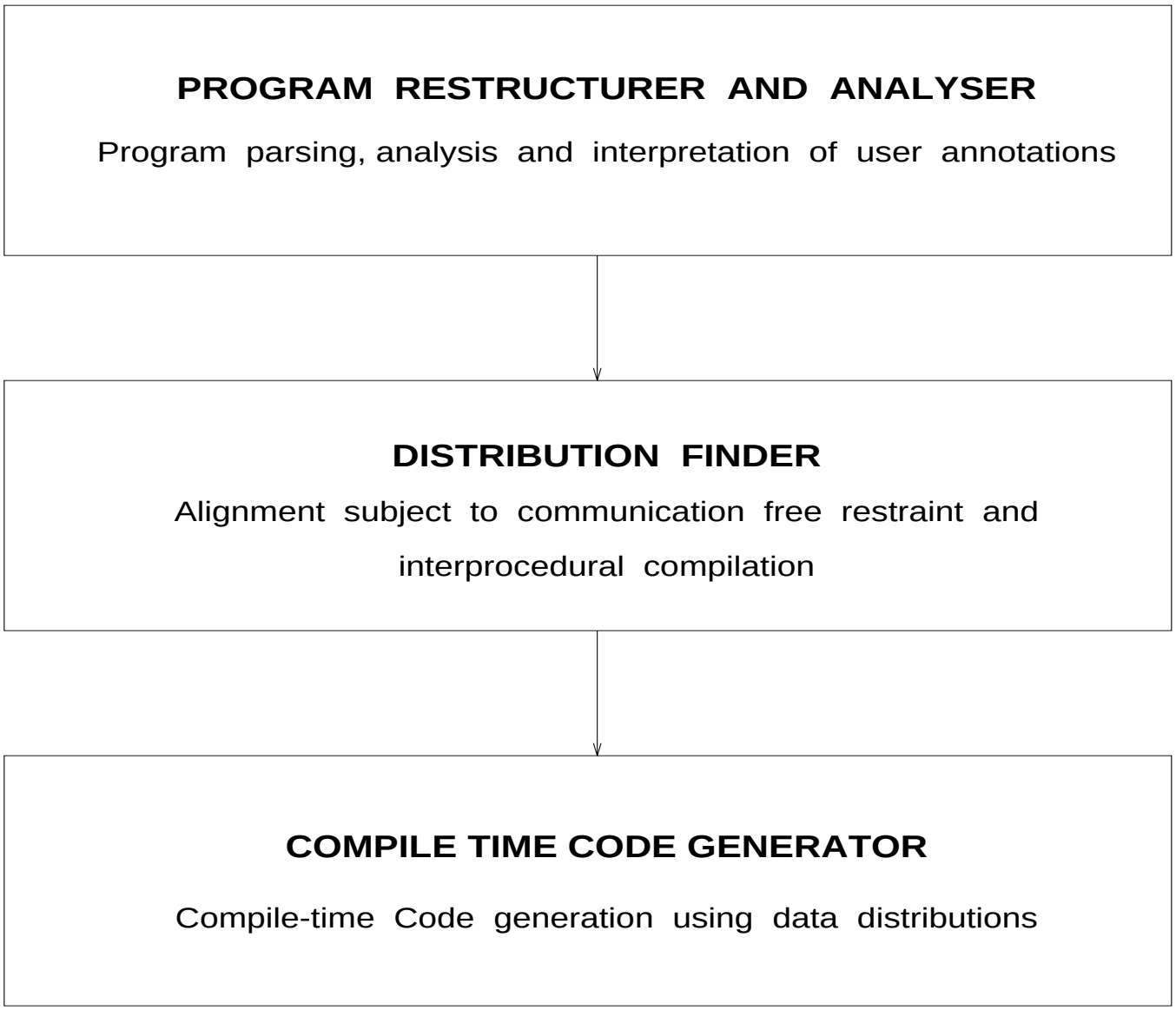

Figure 1.2: System Organization

1.5 Organization of the thesis

We close this chapter by providing an overview of this thesis. Our work involves three major functions : *program restructuring and analyzing*, *determining distributions* and *efficient code generation.*(see figure 1.2) The organization of this thesis is as follows. Chapters 2 to 4 are concerned with the purpose of program restructuring and analyzing. Chapter 2 describes the *Fortran D* language that we have used for the description of the distribution functions in the parallel program. Chapter 3 gives a brief description to *SIGMA II*, the program restructurer and analyser that we have built our system upon. Chapter 4 details the *user annotations* that are currently required by our system. Chapter 5 describes the underlying theory for *alignment and detection of communication* (our second major function). Chapter 6 describes the underlying theory for the parallelization process (the third major function). Finally, chapter 7 concludes with some observations on the future directions on which work should be done.

Chapter 2

Fortran D Language

Fortran D [9] is a version of Fortran enhanced with data-decomposition specifications. It is designed to support two fundamental stages of writing a data-parallel program : *problem mapping* using sophisticated array alignments, and *machine mapping* through a rich set of data distributions functions. We believe that Fortran D provides a simple machine-independent programming model for most data-parallel computations. We have decided to use Fortran-D for describing the distributions in our *information file* as we believe that Fortran D is both powerful and easy enough for the user to understand and use effectively.

2.1 The Data-Parallel Programming Model

The data-decomposition problem can be approached by noting that there are two levels of parallelism in data-parallel applications. First there is the question of how arrays should be *aligned* with respect to one another, both within and across array dimensions. This is called the *problem mapping* problem [9] induced by the structure of underlying computation. It represents the minimal requirements for reducing data movement for the program, and is largely independent of machine considerations. The alignment of arrays in the program depends on the natural fine-grain parallelism defined by individual members of data arrays.

Second, there is the question of how arrays should be *distributed* onto the actual parallel machines. This is the *machine mapping* problem caused by translating the

problem onto the finite resources of the machine. It is dependent on the topology, communication mechanisms, size of local memory and number of processors in the underlying machine. Data distribution provides opportunities to reduce data movement. The distribution of arrays in the program depends on the coarse-grain parallelism defined by the physical parallel machine.

Fortran D requires the user to specify data decomposition in terms of these two levels of data parallelism. First, the ALIGN statement is used to describe a problem mapping. Second, the DISTRIBUTE statement is used to map the problem and its associated arrays to the physical machine.

2.2 DIMENSION Statement

The DIMENSION statement may be used to declare an array which declares the name, dimensionality and size of array.

DIMENSION SEV(TWOJ1)
DIMENSION AP(LONF2,N)

In this example, SEV is declared as an one-dimensional array of size TWOJ1, with elements indexed from 1 to TWOJ1. AP is a two-dimensional LONF2 x N array (stored in memory in column major fashion.)

2.3 DECOMPOSITION Statement

(see figure 2.2) The DECOMPOSITION statement may be used to declare a name for each problem mapping. Arrays in the program are mapped to the decomposition with the ALIGN statement. The result represents an abstract high level specification of the fine-grain parallelism of a problem. There may be multiple decompositions representing different problem mappings, but an array may be mapped to only one decomposition at a time.

Decompositions are designed to enable users to easily group data arrays associated with solving a single problem. The decomposition statement declares the name, dimensionality and size of a decomposition for later use.

DECOMPOSITION SEVTEMP(TWOJ1)
DECOMPOSITION FLNTEMP(LNT2,N)

In this example, SEVTEMP is declared as an one-dimensional decomposition of size TWOJ1, with elements indexed from 1 to TWOJ1. FLNTEMP is a two-dimensional LNT2 x N decomposition.

2.4 LINEARIZED Statement

The LINEARIZED statement may be used to declare those dimensions of an array which have been collapsed for storage efficiency. This statement declares the dimension(s) that has(have) been collapsed, the number of dimensions that have been collapsed into given dimension and section information.

LINEARIZED QLN(DIM = 1, NDIM = 2, FDIM = 162, SECTIONS : 81(0,-2,1))
LINEARIZED VLN(DIM = 1, NDIM = 2, FDIM = 162, SECTIONS : 1(0,0,1),81(0,-2,1))

In this example, QLN is declared as a linearized array with first dimension consisting of two collapsed dimensions, with size of first collapsed dimension as 162 and second collapsed dimension as 81. VLN is a linearized array with first dimension consisting of two collapsed dimensions, with first collapsed dimension size as 162 and second collapsed dimension size as 82.

The precise shape of the arrays are shown in figure 2.1.

2.5 ALIGN Statement

(see figure 2.2) The ALIGN statement is used to map arrays with respect to a decomposition. Arrays mapped to the same decomposition are automatically aligned with each other. Alignment can take place either within or between dimensions.

The alignment of arrays to decomposition is specified by placeholders in the subscript expressions of both the array decompositions. I, J, K, etc. ...are canonical placeholders indicating the location of dimensions in the decomposition. Array

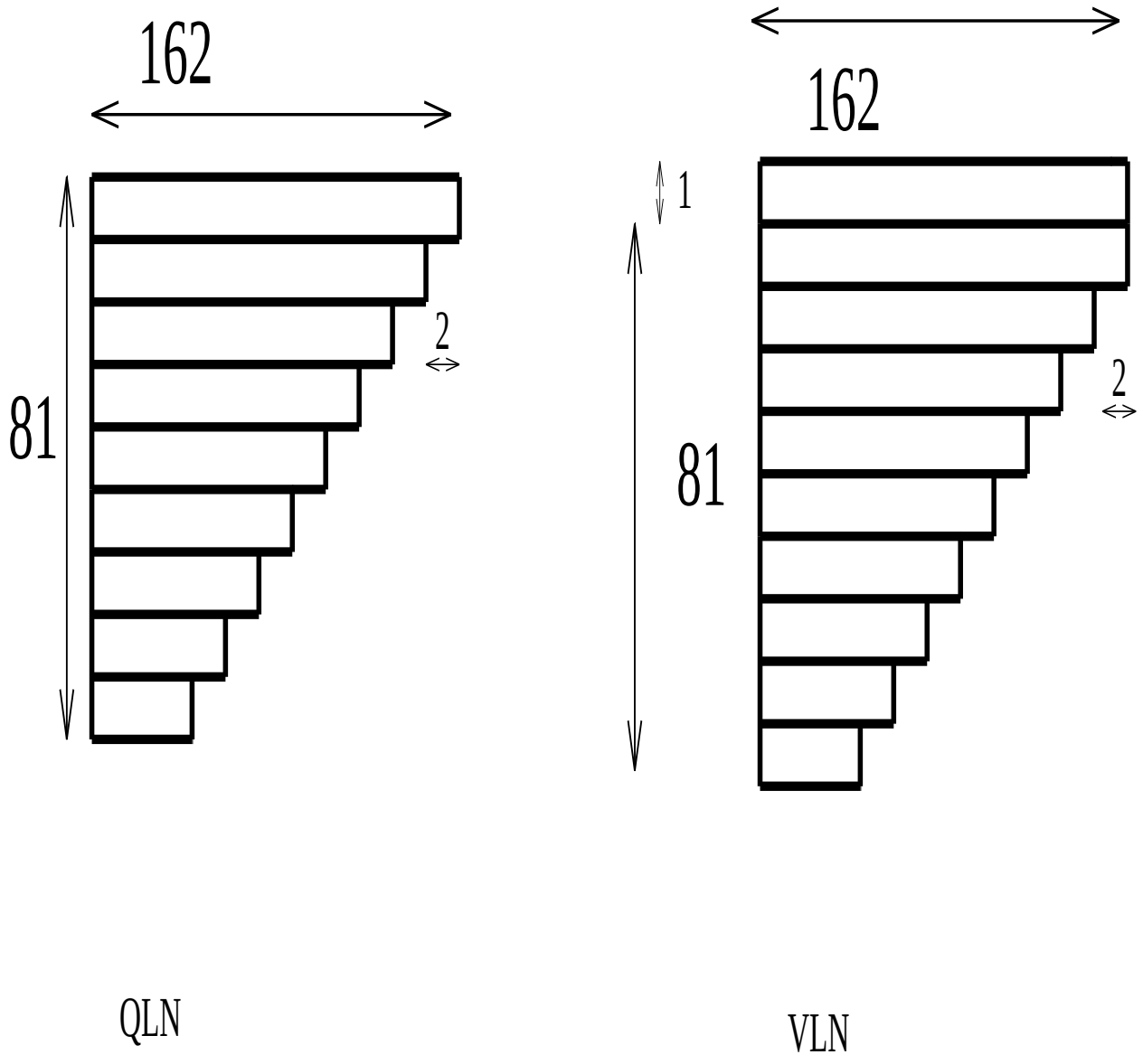

Figure 2.1: Linearized Arrays

subscripts are fixed; they always consist of the placeholders in alphabetical order beginning with I. The decompositions subscripts can be functions of the placeholders; they specify the alignment of array with respect to the decomposition. The array which is aligned with a decomposition whose some dimensions are declared to be linearized is automatically taken to be linearized for corresponding dimensions.

2.5.1 Exact match

The simplest alignment occurs when the array is exactly mapped onto the decomposition. In the following example, the arrays SEV and SOD are mapped exactly onto the equivalent dimensions in the decompositions SEVTEMP.

```
DIMENSION SEV(TWOJ1), SOD(TWOJ1)
DECOMPOSITION SEVTEMP(TWOJ1)
ALIGN SEV(I,J), SOD(I,J) WITH SEVTEMP(I,J)
```

For convenience, placeholders are not required where the mapping is exact. For instance, the alignments in the previous example could also have been specified with the following syntax.

```
ALIGN SEV, SOD WITH SEVTEMP
```

2.5.2 Intra-dimension alignment

Intra-dimension alignment determines the data decomposition within each dimension. This section describes how offset and stride may be specified.

Alignment offsets

The user can specify an alignment offset for any dimension of an array. Constants are added to the placeholders in the decomposition to indicate the offset in that dimension.

```
ALIGN FRN(I,J) WITH FIRNSTEMP(I-1,J)
```

In this example, FRN is aligned with respect to decomposition FIRNSTEMP by -1.

Alignment strides

Fortran D also allows a stride to be specified when performing intra-dimensional alignment. Alignment strides are used to determine the density of an array mapped to a dimension. They are introduced as coefficients of placeholders in the subscript expressions of decompositions in an ALIGN statement. Strides may also be used in combination with offsets.

ALIGN FRN(I,J) WITH FIRNSTEMP(2*I-1,J)

In this example, array FRN has a stride of 2 with respect to decomposition FIRNSTEMP and is mapped to odd element of FIRNSTEMP.

2.5.3 Inter-dimension alignment

Inter-dimension alignment determines the data decomposition between dimensions. This section describes how permutation, collapse and embedding may be specified.

Permutation

The user can arbitrarily permute the dimensional alignment between arrays and decompositions. Canonical placeholders must be used to mark the aligned dimensions.

ALIGN UFLIP(I,J) WITH UFLOPTEMP(J,I)

In this example, the transpose of UFLIP is mapped to the decomposition UFLOPTEMP.

Collapse

(see figure 2.2) It is sometimes convenient to ignore certain dimensions of the array when mapping an array to a decomposition. All data elements in the unassigned dimensions are collapsed and mapped to the same location in the decomposition. An array dimension may be collapsed in the ALIGN statement simply by excluding its placeholder from the decomposition subscripts.

ALIGN QLN(I,J), FLN(I,J,K) WITH SEVTEMP(I)

In this example the first dimension of QLN is mapped onto the decomposition SEVTEMP. The second dimension of QLN is collapsed and stored on the same processor.

Embedding

Conversely, it may be necessary to map arrays with fewer dimensions onto the decomposition. In these cases it is necessary to specify both the mapping for each dimension of the array and the actual position of the array in the unmapped dimensions of the decomposition. This determines the embedding of the array in the decomposition.

ALIGN SEV(I) WITH QLNTEMP(I,2)
ALIGN SOD(I) WITH QLNTEMP(I,1:81)

In the first example, array SEV is mapped to the first dimension of decomposition QLNTEMP, column. It is necessary to specify the actual column position with a constant or a range for unmapped dimension.

2.6 DISTRIBUTE Statement

(see figure 2.2) In Fortran D, the DISTRIBUTE statement can be used to specify the mapping of the decomposition to the physical parallel machine. The distribution specifies the machine mapping for exactly one decomposition. The compiler then applies the distribution to all the arrays mapped to the decomposition. The user does not need to specify a distribution for each array.

DISTRIBUTE SEV(*attribute*)
DISTRIBUTE AP(*attribute,attribute*)

Each attribute describes the mapping of the data in that dimension of the decomposition. Suppose there are P processors and N elements in the decomposition. The four types of attributes for regular distributions in Fortran D are BLOCK, CYCLIC, BLOCK_CYCLIC and DISTRIBUTED. These distributions can be described as follows :

- BLOCK divides the decomposition into contiguous chunk of size N/P, assign-

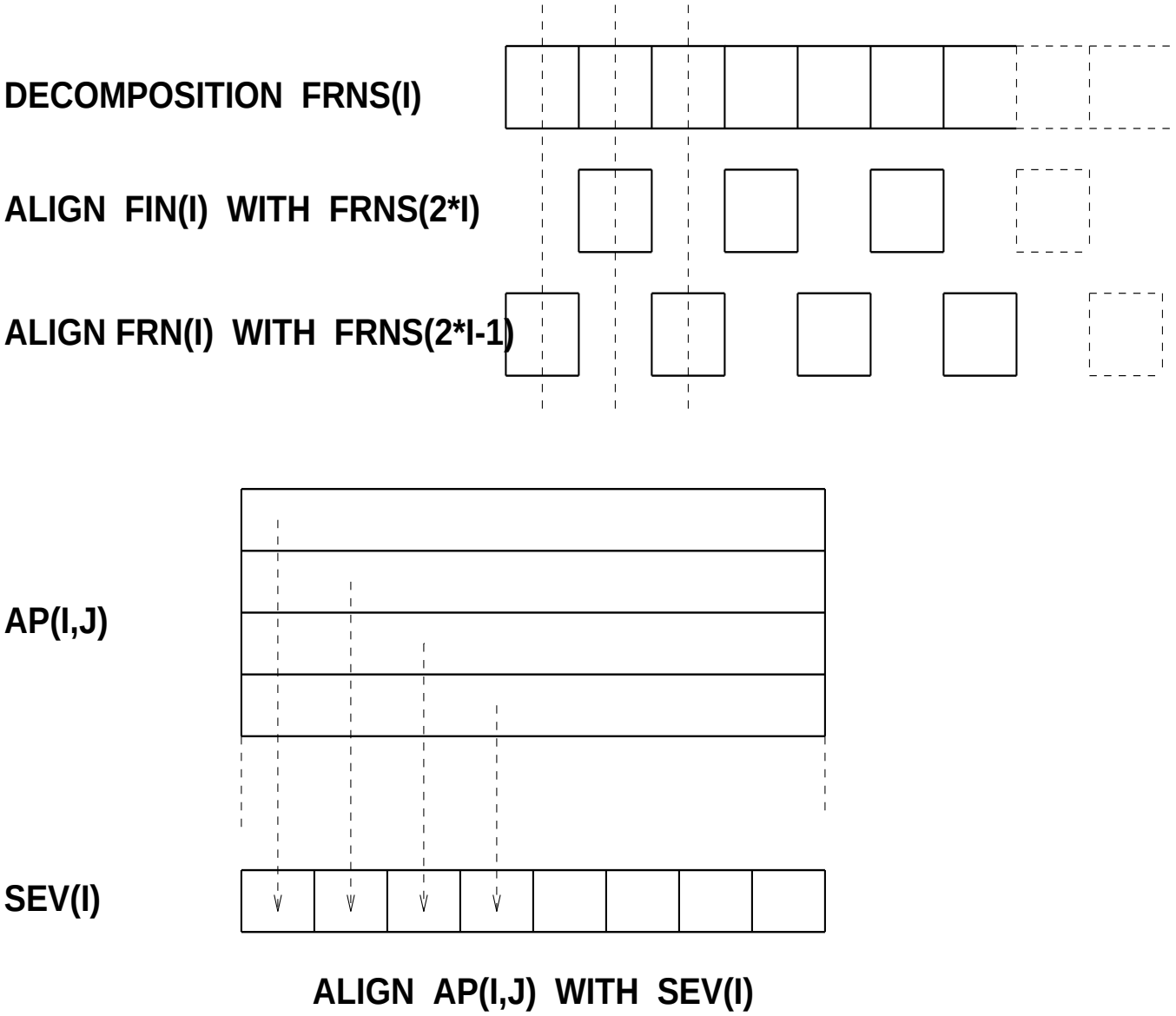

Figure 2.2: Some Fortran D Constructs

ing one block to each processor.

- CYCLIC specifies a round-robin division of decomposition, assigning every P^{the} element to the same processor.
- BLOCK_CYCLIC is similar to CYCLIC but takes a parameter B. It divides the dimension into contiguous chunks of size B, then assigns these chunks in the same fashion as CYCLIC.
- DISTRIBUTED takes parameters O, B and distributes array using the formula

2.7 Discussion

Fortran D is a powerful language for giving a formal and precise specification of the data distributions in a program. It is also simple enough that a sophisticated compiler can produce efficient programs for different parallel architectures. It serves our purpose for specifying the data distributions in the "information file" exactly.

Chapter 3

SIGMA II

SIGMA II [10] is a system for building program transformation tools. Its intended use is for building source to source program parallelization restructurers and performance analysis instrumentation tools for application written in FORTRAN, FORTRAN90, C, C++ and certain extensions of these languages including PCF FORTRAN, Cedar FORTRAN and pC++. SIGMA II contains an interprocedural analysis system and symbolic data dependence procedures. We have built our system on top of SIGMA II. We understood the SIGMA system at the source level and performed additions/modifications according to our needs. Thus for example we have added the grammar of user annotations to the fortran grammar of SIGMA. The user annotations have been detailed in the next chapter. In this chapter we describe the SIGMA system for the purpose of using it effectively for any parallelizing purpose such as program restructuring or analyzing or interpretation of the parse tree built by SIGMA. The discussion found in this chapter is largely based on that in [10] and [1].

3.1 Introduction

SIGMA II provides a tool kit for analyzing and restructuring application programs intended for use on scalable parallel systems. Its intended use is for building end-user tools to aid in the process of program parallelization and performance analysis. Simply put, SIGMA II is a data base for accessing and manipulating program control and data dependence information.

The functions provided by the data base can be used by the end-user tool in one of three ways.

- *Extracting syntactic information :* This includes symbol and type table information, program control flow structure, user annotation and directive information and access to the complete parse tree of the application.
- *Extracting semantic information :* This includes interprocedural definition and use summaries for each function and procedure, data dependence analysis and scalar propagation and symbolic analysis and simplification of scalar expressions.
- *Restructuring* the program by modifying the data case contents and to generating (unparsing) new versions of the source code based on modifications.

3.2 The SIGMA Architecture

The *SIGMA* systems consists of two basic components:

1. A set of parsers one each for FORTRAN, C and pC++ that translate the source code into a special internal form.
2. A library of data base utility for extracting and modifying information stored in the internal form generated by the parsers.

Parallel programming tools are designed to work with user application program which are often very large and exist as multiple source files. In Sigma terms each application program defines a *project* which consists of a set of source files, a set of dependence files and a project file. The source files are in our case Fortran. For each source file there is a corresponding dependence file generated by the parser. (The convention is that source file end in ".f" or ".c" and the corresponding dependence file end in ".dep" with the same root name.) The project file is an ascii file containing a list of all the associated dependence files in the project. (The project file has a ".proj" suffix and the dependence files are listed one by line.)

The dependence file for each source file contains a complete parse tree and a symbol table for the program fragment in that file. In addition it contains a first pass analysis of the interprocedural flow of the functions and subroutines in that module.

The *SIGMA Parsers* are invoked as follows. For FORTRAN programs the command line is

```
cfp  filename.f
```

One way to think of the Sigma data base is as a repository of information about the source code of an application. The data base can be thought of as a heavily annotated parse tree and symbol table. The data base functions are tools to extract information about the program semantics or to modify the program structure. Among the types of annotations supported are

1. Interprocedural information about variable and parameter uses and definitions.
2. Scalar propagation information.
3. Data Dependences and distance/direction vectors. This includes a fairly complete symbolic analysis package for manipulating subscript expressions.
4. Source Annotations defined by a special *assert* comment that is understood by the parsers.
5. User defined annotations supported by a *SiPutProperty(), SiGetProperty()* function pair.

3.2.1 The database structures

In the Sigma model, each program is a collection of graphs and tables corresponding to a collection of source files. Each graph is a structured parse tree for the program. The nodes correspond roughly to FORTRAN statements and the tree structure is based on control constructs. The children of a statement are called the *control children* of a node which is known as the *control parent*. The root of the graph is called the *global node* for the file. The nodes with control children correspond to

- the global node for a file,
- subroutine or function headers,
- looping constructs.

- conditionals ($IF - then - else$).

Because of pure conditionals like *IF-then-else* each set of control nodes has two sets of children, the *false branch* and the *true branch*, but for most control nodes the *false branch* is empty.

In the data base each statement, expression, symbol table and type table entry is identified with an integer. This *node id* is the principle means of access to the information stored with a node. Each statement node has several potential attributes including

- the type of the node which identifies the statement type,
- the name and line number of the source file containing this statement,
- the identifier of the control parent of this node,
- a symbol reference such as a do loop parameter in FORTRAN or a subroutine name in a call statement,
- a list of data dependences associated with the statement,
- the user defined properties,
- the comments that precede the statement (and the associated program annotations),
- two or three expressions associated with the node (see the next sections for details),
- the control children of the node.

Each of these attributes can be accessed for a given statement by one of the data base The data type used by SIGMA II to communicate the structure of the programs in the data base to the end-user applications is called an *ELIST* which is a Lisp type s-expression. All library functions return values which are either integers or pointers to *ELIST* values.

3.2.2 Low-level expressions

A statement node can have up to three subexpressions associated with it. By expression we mean lists of or algebraic expressions of variables, constants or functions. For example, a FORTRAN *do* loop has an index initialization expression, an index termination expressions and an index increment expression. A FORTRAN assignment statement has a left hand side expression and a right hand side expression. (On the other hand, a C assignment statement, called an *ASSIGN_EXP* has only one expression because the assignment operator returns a value and the comma can be used to generate a list of expressions to be evaluated in one *statement* The primitive *get_both_sides()* from the data base provides access to the first two expressions of a statement.

Each Expression node has four components. The first component is the type of the node, the second component is an (optional) symbol reference. The third and fourth components are the *left* and *right* operand expression nodes.

There are many functions that manipulate expressions and data dependence information. The complete list of functions and their definitions is given in the SIGMA documentation [10] and [1].

In addition to the data base of statement and expression trees, Sigma contains a full symbol table that is indexed by the id and the name of the variable. A type table is also provided.

3.3 Discussion

SIGMA II is a tool of substantial value in the construction of a number of parallel programming tools and provides researchers an easy way to access language syntax and semantics. It has been employed extensively by us in our system.

Chapter 4

User Annotations

User annotations in the sequential fortran program may be used to aid the compiler in generation of correct and efficient compile-time code. They may be used by the compiler to deduce data decompositions of some arrays used in the code. All the annotations start with reserved word *cparl$*. Here we give the description of each user annotation along with the example from the weather code and the particular routine where that particular annotation has been employed. These user annotations have been integrated with the SIGMA parser and the useful semantic information extracted from them is written in the dependence file generated by the SIGMA system to be later read and made use of by the distribution finder and the code generator. The lexical analyzer of SIGMA (fortran lexer) has also been altered in order to allow for these special user comments.

4.1 DISTRIBUTION FILE IS Statement

The DISTRIBUTION FILE IS statement may be used to declare an information file which contains data decomposition information of the arrays used in the code in the Fortran D language (see Chapter 2.)

cparl$ DISTRIBUTION FILE IS "sums2a.info"

In this example, *sums2a.info* is declared as a file containing data decomposition information of the arrays used in the code.

4.2 IDENTICAL Statement

The IDENTICAL statement may be used to declare the fact that some array in the code has identical data decomposition as an array in distribution file. This may be required if the dimensions and data access patterns of some array in this code is identical to an array variable in the code whose distribution file is specified in previous annotation.

cparl$ DISTRIBUTION FILE IS "sums2a.info"
cparl$ IDENTICAL FLN,VLN

In this example, VLN is declared to have an identical data decomposition as FLN in distribution file *sums2a.info*.

4.3 SIMILAR Statement

The SIMILAR statement may be used to declare the fact that some array in the code has some similarity in decomposition as an array in distribution file. This may be required if the data access patterns of some array with respect to some dimensions in this code is identical to an array variable in the code whose distribution file is specified in previous annotation.

cparl$ DISTRIBUTION FILE IS "sums2a.info"
cparl$ SIMILAR FLN,VLN EXCEPT SIZE
cparl$ SIMILAR [1,2]AP,[2,1]QLN

In the first example, VLN is declared to have an identical data decomposition as FLN in distribution file *sums2a.info* except that it has different size. In the second example second and first dimensions of QLN are declared to have identical distributions as first and second dimensions of AP respectively.

4.4 LINEARIZED Statement

The LINEARIZED statement may be used to declare those dimensions of an array which have been collapsed for storage efficiency. This statement declares the dimension(s) that has(have) been collapsed, the number of dimensions that have been

collapsed into given dimension and section information.

cparl$ LINEARIZED QLN[DIM = 1, NDIM = 2, FIRSTSIZE = 162, SECTIONS = 81(0,-2,1)]

In this example, QLN is declared as a linearized array with first dimension as collapsed (we assume that when a dimension is collapsed it consists of two dimensions), with size of first collapsed dimension as 162 and second collapsed dimension as 81. The shape of QLN is same as that in figure 2.1. The reason for including this annotation is that the exact shape of the linearized array is recorded by our program analyzer. When a reference occurs in the form of a specific access pattern of a particular linearized array, we first *delinearize* it i.e. we compute the access pattern relative to the array that is obtained if we assume that the dimensions of the linearized array were not collapsed,

4.4.1 Delinearization

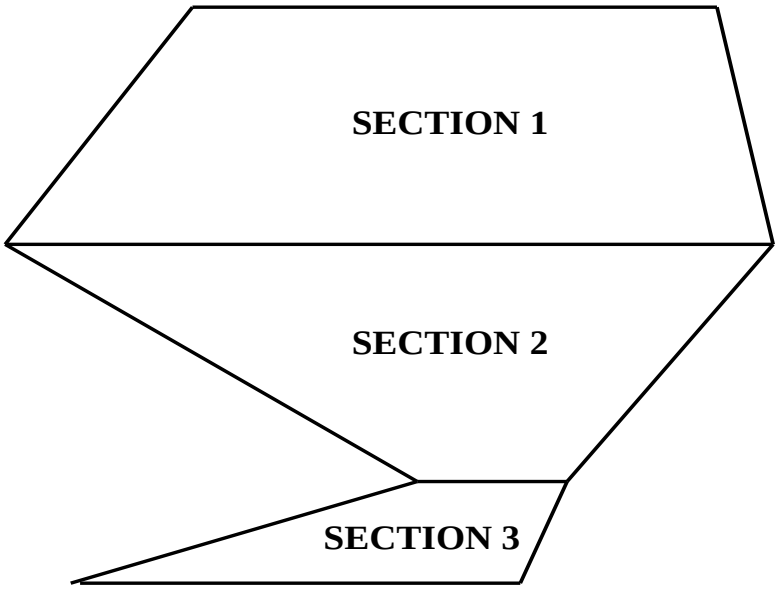

Figure 4.1: A Linearized Array composed of several L-sections

The basic element used for delinearization is a trapezoid. A linearized array of complicated shape is broken into a series of trapezoidal sections(see figure 4.1). Suppose QLN is a linearized array with two dimensions in the linearized form and one dimension in the delinearized form. Then our goal is to transform an expression QLN(ex)

which refers to some element of the linearized array to an equivalent expression of the form QLN(exi,exj) which will also refer to the same element in the delinearized form of QLN. Consider an L-section (a linearized array is composed of many trapezoidal L-sections) for a linearized array (see figure 4.2). The parameters completely specifying this L-section are : 1) f, the size of the first strip 2) h, the height of this section 3) the left and the right skips l and r respectively (which are constant for a trapezoidal section) and 4) the stride s which indicates after the height of one strip. (The L-section is composed of many strips, the last one of which may be of height less than s.) Also let the linearized array be mapped to a two-dimensional array of size $A * B$ where B is the sum of heights of all the sections, and A is the distance between the leftmost and the rightmost element in the linearized form of array.

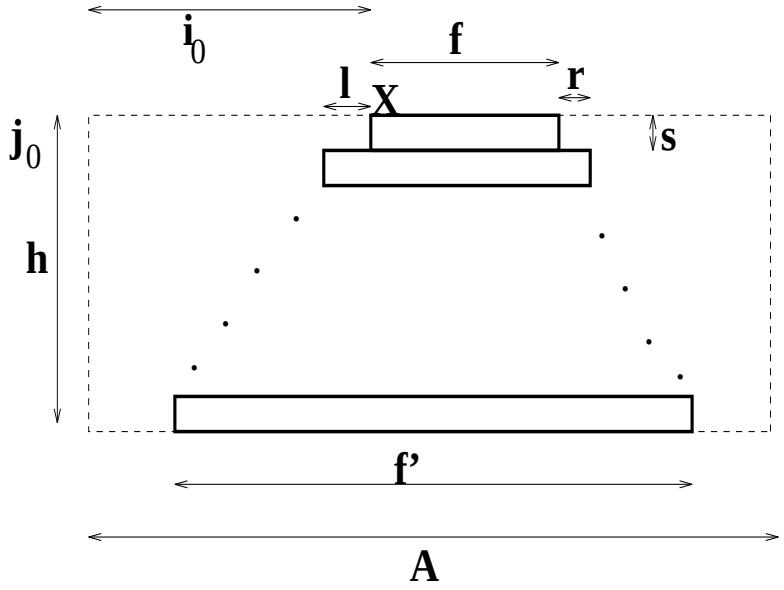

Figure 4.2: An L-section of a Linearized Array

Consider an index expression (I, J) of the delinearized array (see figure 4.2). We will calculate what this expression will be equivalent to in the one-dimensional linearized array. Number of complete strips $= \lfloor \frac{h-1}{s} \rfloor = t$, with the last strip of height

$$h - s\lfloor \frac{h-1}{s} \rfloor = (h-1) \pmod{s} + 1$$

The length of the last strip is $f' = f + (l+r) * (\lfloor \frac{h-1}{s} \rfloor)$ Let the point X in the figure 4.2 be (i_0, j_0). Then we have $l_k = l(k-1)$ and $r_k = r(k-1)\ \forall 1 \leq k \leq (t+1)$ where k denotes the strip number and l_k and r_k denote the displacement of the leftmost

and rightmost points of the k^{th} strip from the beginning strip (strip number j_0) of this L-section. The element (I, J) say P lies in the strip number $\lfloor \frac{J-j_0}{s} \rfloor + 1$. First we count the number of elements in $c = \lfloor \frac{J-j_0}{s} \rfloor$ (complete) strips. This is given by the sum of an arithmetic progression as

$$N_c = s \sum_{k=1}^{c} \{f + (l+r)(k-1)\} = s\{fc + (l+r)c(c-1)/2\}$$

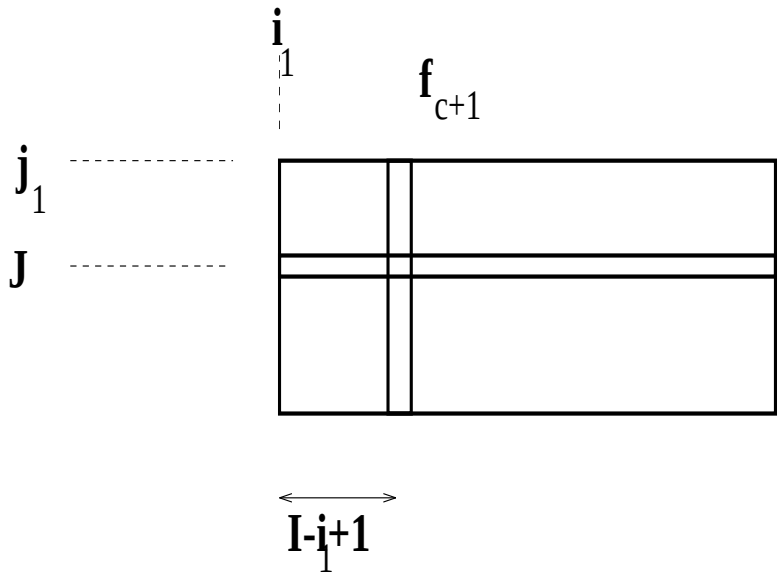

Figure 4.3: The $(c+1)^{st}$ Strip

Now the $(c+1)^{st}$ strip is as shown in figure 4.3 Then $P(I, J)$ corresponds to $X + N_c + (J - j_1)f_{c+1} + (I - i_1)$ with $f_{c+1} = f + (l+r)c$, $j_1 = j_0 + cs$ and $i_1 = i_0 - lc$ Putting these values and simplifying, we get that the point P refers to the following element in this L-section:

$$X + (I - i_0 + lc) + (J - j_0)(f + (l+r)c) - (l+r)c(c+1)s/2$$

where X is the number of elements in the array before the start of this section (X = 1 for the first section). Now given any access pattern we match the coefficients of the subscript expression with the above obtained expression. More specifically, taking the assumption that the subscript J moves in skips of s (i.e., that no more than one strip of any L-section is referred to in one pass of the L-section) we get the coefficients in terms of the two enclosing loop induction variables (with $I = Qi + R$ and $J = sMj + N + j_0$:

$$(\text{coefficient of } j^2) : s(l+r)M/2$$

$$(\text{coefficient of } j) : M * (l + sf + N(l + r) - s(l + r)/2)$$

$$(\text{coefficient of } i) : Q$$

$$(\text{constant coefficient: }) : X + R - i_0 + l * ns + b(f + (l + r)ns) - s(l + r)ns(ns + 1)/2$$

where ns is $\lfloor N/s \rfloor$ So we equate these coefficients with those of the used subscript expression in the linearized array to solve for the parameters M, N, Q, R and obtain the equivalent delinearized pair of indices (I, J).

4.5 TREAT AS NORMAL Statement

The TREAT AS NORMAL statement is used to specify that a given array whose dimension has been collapsed is being treated as normal between the specified labels.

cparl$ LINEARIZED QLN[DIM = 1, NDIM = 2, FIRSTSIZE = 162, SECTIONS = 81(0,-2,1)]
cparl$ 10,20 TREAT LINEARIZED QLN AS NORMAL

In above example, array QLN whose first dimension has been collapsed is being used as a normal array between the labels 10 and 20. When such a comment is seen, we do not delinearize the particular linearized array i.e. it is treated as normal between the specified labels.

4.6 RECURRENCE Statement

The RECURRENCE statement is used to specify that fact that some variable is dependent on induction variables between the specified labels with the dependence as specified in the annotation. This dependence may be *run-time* or *compile-time*.

4.6.1 Compile-time recursion

The user may specify that given variable is dependent on some induction variable but the dependence can be determined at compile-time. This is particularly useful in the access of linearized arrays which are accessed primarily using recursion variables.

cparl$ 10,20 RECURRENCE IS IPLUS = 322 + (322 - 4*j)*(j - 1)

In this example, variable IPLUS is declared to have compile-time dependence on induction variable on j between the labels 10 and 20. The dependence is given by the expression $322 + (322 - 4 * j) * (j - 1)$.

4.6.2 Run-time recursion

The user may specify that a variable has run-time dependence on some induction variable.

```
cparl$ 10,20 RECURRENCE IS PROD : RUNTIME
          PROD = PROD * Y(L)
```

In this example variable PROD is declared to have a run-time dependence on the induction variable L whose loop starts at label 10. This annotation is required because runtime recursions cannot be normally handled by compile-time resolution and different strategy has to be taken in such cases.

4.6.3 An extended example for the linearized and recurrence statements

Consider the following example from a section of the code from a routine "sums2a.f" used in "pgloopa.f".

```
 cparl$ linearized QLN [NDIM = 2,FIRSTSIZE = 162,SECTIONS =
81(0,-2,1)]
 cparl$ linearized FLN [ NDIM = 3,FIRSTSIZE = 162,SECTIONS =
81(0,-2,1)]


 IPLUS = TWOJ1*2 - 2
LEN = TWOJ1 - 4
cparl$ 31,40 recurrence is IPLUS = 322 + (322 - 4*j)*(j - 1)
cparl$ 31,40 recurrence is LEN = 162 - 4*j
31 DO 60 J=1,NPAIR
DO 40 I=1,LEN
SEV(I) = SEV(I) + QLN(I+IPLUS) * FLN(I+IPLUS,K)
40 CONTINUE
```

```
IPLUS = IPLUS + LEN
LEN = LEN - 2
cparl$ 41,50 recurrence is IPLUS = 480 + (318 - 4*j)*(j - 1)
cparl$ 41,50 recurrence is LEN = 160 - 4*j
41 DO 50 I=1,LEN
SOD(I) = SOD(I) + QLN(I+IPLUS) * FLN(I+IPLUS,K)
50 CONTINUE
IPLUS = IPLUS + LEN
LEN = LEN - 2
60 CONTINUE
```

It is clear from the recurrences of iplus and len that the delinearized form of the access QLN(I+IPLUS) is QLN(I,J). It is this linearized form that is used for alignment and code generation purposes.

4.7 REDUCE Statement

A reduction is an operation on a collection of data that results in new data of lesser dimensionality, usually a single scalar value. User annotations provide the REDUCE statement as an optional method of specifying reductions that the computer may find difficult to detect.

```
cparl$ REDUCE(RESULT : SEV; SIMPLE QLN; LINEARIZED FLN)
cparl$ REDUCE(RESULT : FUNEV(I); SIMPLE PLN(^ ),UFLIP(I,^
); LINEARIZED )
```

The first example declares that some dimension of QLN and FLN are being reduced to produce the final result in array SEV. The second example declares that first dimension of PLN and second dimension of UFLIP are being reduced to produce result in FUNEV. In addition first dimension of FUNEV is aligned with first dimension of UFLIP.

4.8 Discussion

User annotations help the compiler to generate correct compile-time code. They also help the distribution finder to figure out distributions of certain arrays. While

it would be ideal to be possible to achieve the same purpose without any user annotations, the fact remains that these user annotations are valuable in aiding program analysis and reducing the execution time of our system.

Chapter 5

Determination of Data Distributions

In this chapter we will discuss the main issues involved in the determination of unknown data decompositions. When modifications are made to sequential code and the system has to generate the parallel code, the primary task is to determine data distributions of the new array variables which have been introduced in the modification of the sequential code. This is achieved by their *alignment* with those variables whose distributions are known i.e., were described in the *information file*. After alignment is carried out, we first check whether the modifications made to the sequential code would entail communication in the parallel code. As we will see in this chapter the two problems : *alignment* and *detection of communication* are closely related.

5.1 Data Distribution

In this section we describe our abstract machine and the kind of distributions arrays may have in our scheme. The abstract target machine we assume is a D-dimensional (D is the maximum dimensionality of any array used in the program) grid of $N_1 * N_2 * N_3 * \ldots N_D$ processors. Such a toplogy can easily be embedded on almost any distributed memory machine. A processor in such a topology is represented by the tuple $(p_1, p_2, \ldots, p_D), 0 \leq p_k \leq N_k - 1$ for $1 \leq k \leq D$. The correspondence between a tuple $(p_1, p_2, \ldots, p_D)$ and the processor number in the range 0 to $N - 1$

is established in the scheme which embeds the virtual processsor grid topology on the real target machine. We have adapted the terminology of [11] for the data distribution scheme. The representation is extended to make the *replication* of data simpler to represent in this scheme. A processor tuple with an X appearing in the i^{th} position denotes the processors along the i^{th} grid dimension. Thus for a $2x2$ grid of processors, the tuple $(0, X)$ represents the processors $(0, 0)$ and $(0, 1)$, while the tuple (X, X) represents all the four processors.

The scalar variables used in the program are assumed to be replicated on all processsors. For other arrays a separate distribution function is used with each dimension to indicate how that array is distributed across processsors. The k^{th} dimension of an array A is referred to as A_k. Each dimension A_k is mapped to a unique dimension $map(A_k), 1 \leq map(A_k) \leq D$, of the processor grid. If $N_{map(A_k)}$, the number of processors along that grid dimension is 0, we say that array dimension A_k has been sequentialized. The sequentialization of an array dimension implies that all elements whose subscripts differ only in that dimension are allocated to the same processor. The distribution function for A_k takes as its argument an index i and returns the component $map(A_k)$ of the tuple representing the processor which *owns* the element $A[-, -, \ldots, i, \ldots -]$, where " $-$" denotes an arbitrary value, and i is the index appearing in the k^{th} dimension. The array dimension A_k may either be partitioned or replicated on the corresponding grid dimension. The distribution function is of the form:

$$f_A^k(i) = \lfloor \frac{i - o}{b} \rfloor [\quad (\text{mod } N_{map(A_k)})]$$

if A_k is partitioned ; and is denoted X if A_k is replicated. The square parentheses surrounding mod $N_{map(A_k)}$ indicate that the appearance of this part in the expression is optional. At a higher level, the given formulation of the distribution function can be thought of as specifying the following parameters: 1) whether the array dimension is partitioned across processors or replicated, 2) method of partitioning –contiguous or cyclic, 3) the grid dimension to which the k^{th} array dimension gets mapped, 4) the block size for distribution, i.e., the number of elements residing together as a block on a processor, and 5) the displacement applied to the subscript value for mapping.

Examples of some data distribution schemes possible for a $16x16$ array on a four-processor machine are shown in figure 5.1. The numbers shown in the figure indicate the processor(s) to which that part of the array is allocated. The machine is con-

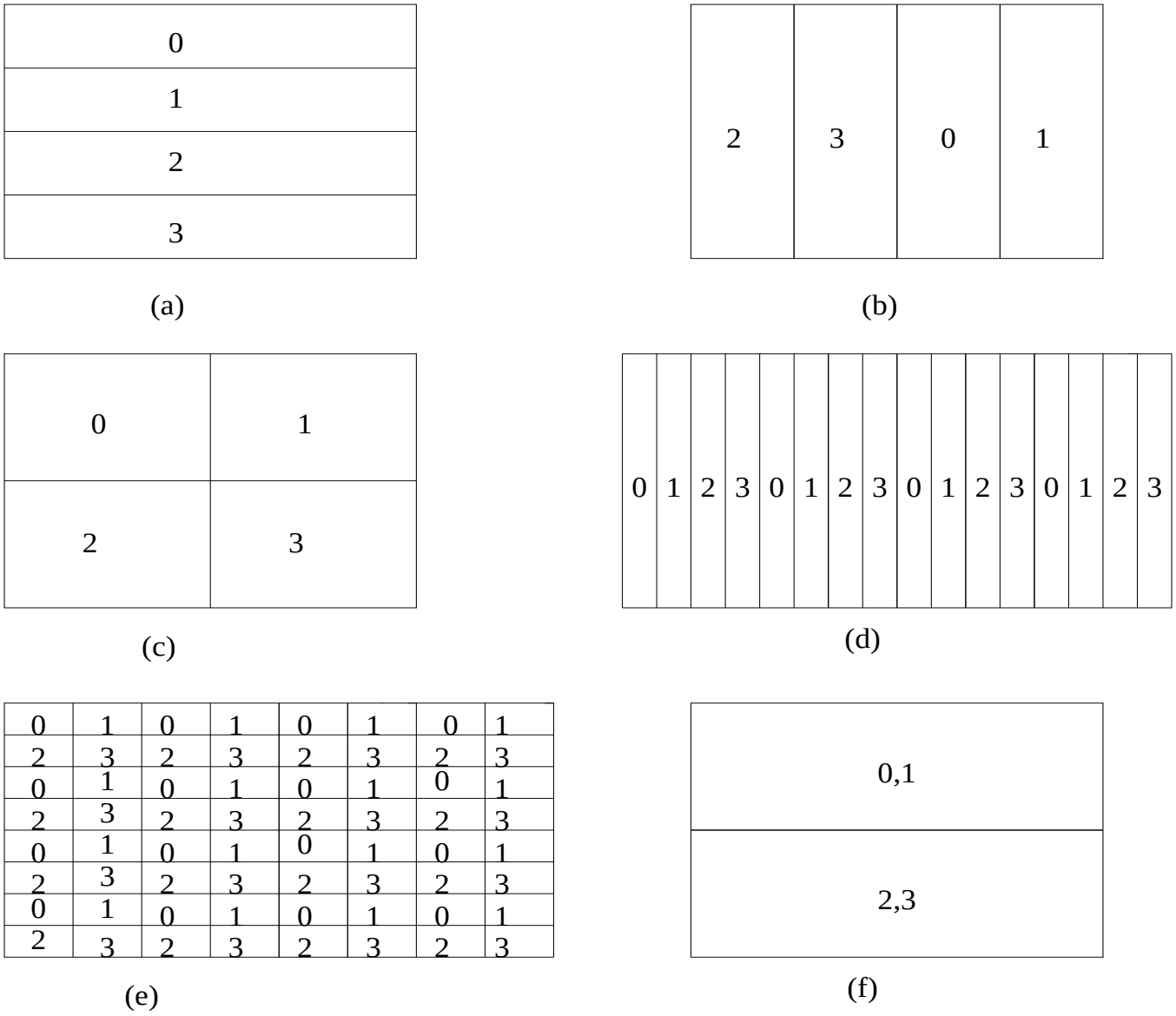

Figure 5.1: Different data partitions for a 16 * 16 array

sidered to be an $N_1 x N_2$ mesh, and the processor number corresponding to the tuple (p_1, p_2) is given by $p_1 * N_2 + p_2$. The distribution functions corresponding to the different figures are given below. The array subscripts are assumed to start with the value 1, as in Fortran.

$$a) N_1 = 4, N_2 = 1 : f_A^1(i) = \lfloor \frac{i-1}{4} \rfloor, f_A^2(j) = 0$$

$$b) N_1 = 1, N_2 = 4 : f_A^1(i) = 0, f_A^2(j) = \lfloor \frac{j-1}{4} \rfloor$$

$$c) N_1 = 2, N_2 = 2 : f_A^1(i) = \lfloor \frac{i-1}{8} \rfloor, f_A^2(j) = \lfloor \frac{j-1}{8} \rfloor$$

$$d) N_1 = 1, N_2 = 4 : f_A^1(i) = 0 f_A^2(j) = (j-1) \pmod 4$$

$$e) N_1 = 2, N_2 = 2 : f_A^1(i) = \lfloor \frac{i-1}{2} \rfloor \pmod 2, f_A^2(j) = \lfloor \frac{j-1}{2} \rfloor \pmod 2$$

$$f) N_1 = 2, N_2 = 2 : f_A^1(i) = \lfloor \frac{i-1}{8}, f_A^2(j) = X$$

The last example illustrates how our notation (taken from [11] allows us to specify partial replication of data, i.e., replication of an array dimension along a specific dimension of the processor grid. If the distribution function for each of its dimensions takes the value X, then the array is completely replicated on all the processors.

Most of the arrays used in real scientific programs and all of the arrays used in the weather code have atmost three dimensions. And hence we assume that our underlying topology is a *three-dimensional mesh.*

5.2 Alignment and Detection of Communication Requirements

Having introduced our notation for internally specifying data distributions, we now present the theoretical issues involved in alignment and checking whether a particular statement can be executed in parallel without communication between processors. Consider the following statement:
for $i = L$ to U by S do :

$$A(g(i)) \sim B(h(i))$$

The communication detection problem requires us to verify that the above loop can be executed without any communication requirement i.e., both the arrays A and

B get mapped to the same processor for all the subscript values occurring in the loop. The alignment problem is only slightly different from this. In the alignment problem, we do not know the distributions of either A or B or both, and based on the relation between the subscript functions $g(i)$ and $h(i)$ we have to deduce their distributions, or record any relation between their distributions so that if at any later point of time we are able to determine the distribution of one array, we can determine the distribution of the other using the determined relation. Since we have specified the distributions in terms of three parameters, *grid dimension, offset and block size*, we have to carry out three alignments axis,offset and stride.

5.2.1 Alignment

Existing Methods

The problem of determining the alignment of dimensions of various arrays has been referred to as the *component alignment* problem by [12]. They prove the problem NP-complete and give an efficient heuristic algorithm for it. In their approach, an undirected, weighted graph called a *component affinity graph* (CAG) is constructed from the source program. For every constraint on the alignment of two dimensions, an edge having a weight equal to the quality measure of the constraint is generated between the corresponding two nodes. The component alignment problem is defined as partitioning the node set of the CAG into D (D being the maximum dimension of arrays) disjoint subsets so that the total weight of edges across nodes in different subsets is minimized, with the restriction that no two nodes corresponding to the same array are in the same subset. Thus, the (approximate) solution to the component alignment alignment problem indicates which dimensions of various arrays should be aligned. A one to one correspondence between each class of aligned array dimensions and a virtual dimension of the processor grid topology can then be established. However we do not consider this algorithm to be suitable for our purpose, mainly because it performs only axis alignment and the weather code has very few, if any, instances where the dimension correspondence between two arrays is not same. The main issue in alignment of variables used in the weather code is that of offset and stride alignment, and these issues are largely not dealt by the component alignment algorithm of [12].

Recently another sophisticated alignment algorithm was proposed by Chatterjee et.

al. in [6] where they consider the issue of automatic array alignment in data-parallel programs. They also formulate the alignment problem as a constrained optimization problem of the residual communication cost function. They build a DAG from the source program and give a dynamic programming algorithm to solve their optimization problem. However, the algorithm has the restriction that a new temporary has to be generated for each occurrence of an array variable on the left hand side. This is clearly a major problem in our case where the same array has been assigned to a large number of times (in many routines of the weather code.) Moreover their algorithm is designed to be efficient when there are communications involved and the algorithm thus incurs considerable overhead in order to perform optimization of communications. In our case there are no communications, so an algorithm that spends major effort in optimization of communications would not suit our purpose.

Our algorithm

We basically follow two approaches towards the alignment problem.

- Solving $\forall i = L, \ldots, U$ by S

$$\lfloor \frac{g(i) - o_A}{b_A} \rfloor \pmod{N_A} = \lfloor \frac{h(i) - o_B}{b_B} \rfloor \pmod{N_B}$$

- Solving $\forall p$

$$g^{-1}(image_A(p)) \cap [L : U : S] = h^{-1}(image_B(p)) \cap [L : U : S]$$

The first approach attempts to solve the equation obtained when we replace the left and right sides of the statement in a loop by their corresponding distribution formulations. By a careful analysis of the various cases of the functions $g(i)$ and $h(i)$ we can derive relations between the parameters $b_A \& b_B$, $N_A \& N_B$ and $o_A \& o_B$. If the form of the functions $g(i)$ or $h(i)$ is non-linear then this approach fails to derive any useful relation between the distribution of the arrays. As an example, when $N_A = N_B$, then one solution is $b_A = b_B$ and $o_A - o_B = 0 \pmod{b_A * N_A}$ in the case when the functions g and h are identity functions.

In the second approach we attempt to equate the *iteration sets* of each processor in the given loop. (see next chapter for a detailed discussion of *image* function

and *iteration sets*) The usefulness of this approach is that it simulatneously eases the task of compile-time code generation because the local iteration sets computed in this approach are used in the loop bounds reduction procedure of compile-time resolution.

Our scheme for array alignment is an integrated form of the above two approaches. The basic idea in our strategy is to consider all constraints of various arrays indicated by the important components of the program, and combine them in a consistent manner to obtain the overall data distribution. It is assumed that there are no conflicts between mutually inconsistent constraints, since there is no communication in the parallel code to be generated. For example consider the loop shown below:
for $i = 1$ to n by 1 do :

$$A(i, c_1) \sim \mathcal{F}(B(c_2, i))$$

The data references in this loop suggest that A_1 should be aligned with B_1, and A_2 should be sequentialized. Second, they suggest the following distribution function for B_1, in terms that for A_1.

$$f_B^1(c_2 * i) = f_A^1(i) \text{ or } f_B^1(i) = f_A^1(\lfloor i/c_2 \rfloor)$$

Thus given parameters regarding the distribution of A, like the block size, the offset, and the number of processors, we can determine the corresponding parameters regarding the distribution of B by looking at the relationship between the two distributions.

5.2.2 Detection of communication requirements

Detecting whether a particular statement inside a parallelizable loop can be executed on the same processor is an easier problem than alignment, since in this problem we know the distributions of the arrays on both sides of the assignment and we have to verify whether there is any communication generated in that statement. Clearly, if the statement is such that if it was considered while alignment was being carried out, then it would not require any further consideration, since the fact that no communication is generated has already been taken account while carrying out the alignment procedure. However, when this statement was not considered while alignment was carried out, it has to be considered here. Suppose we have

$$A(ai + b) \sim B(ci + d)$$

to be considered for detection of communication, where i is the loop variable. Suppose the distribution parameters of A are $< o_A, b_A, N >$ and those of B are $< o_B, b_B, N >$. Then if a divides b_A and c divides b_B, the equation:

$$\lfloor \frac{ai + b - o_A}{b_A} \rfloor \pmod{N} = \lfloor \frac{ci + d - o_B}{b_B} \rfloor \pmod{N}$$

becomes equivalent to solving the equation:

$$\lfloor \frac{i - o'_A}{b'_A} \rfloor \pmod{N} = \lfloor \frac{i - o'_B}{b'_B} \rfloor \pmod{N}$$

where

$$o'_A = \lceil \frac{o_A - b}{a} \rceil$$
$$b'_A = b_A/a$$
$$o'_B = \lceil \frac{o_B - d}{c} \rceil$$
$$b'_B = b_B/c$$

and now the equation can be easily solved by equating the corresponding offsets and block sizes (the newly computed ones.)

5.2.3 Solving the detection of communication problem precisely

The distribution function $f_A(i)$ of an array reference $A(q(i))$ is referred to as $f(i, q, o, b, n) = \lfloor \frac{q(i)-o}{b} \rfloor \pmod{n}$ We will solve in this subsection various particular cases of the problem :

$$\text{given that} \lfloor \frac{q_A(i) - o_A}{b_A} \rfloor \pmod{n_A} = \lfloor \frac{q_B(i) - o_B}{b_B} \rfloor \pmod{n_B} \; \forall i \in \{L \ldots U\} \tag{5.1}$$

what is the relationship between the parameters o_A, o_B ; b_A, b_B and n_A, n_B. In most the cases discussed here $n_A = n_B = n$ and the subscript functions $q_A(i)$ and $q_B(i)$ are linear in i.

In many of the cases mentioned below we will use the following well-known fact:

Let g denote the gcd of a and m i.e (a, m). Then the equation $ax \equiv b \pmod{m}$ has no solutions if g does not divide b. If $g|b$ (i.e. g divides b), it has g solutions : $x = bx_0/g + t(m/g) \pmod{m}$ for $t = 0, 1, \ldots, g-1$ where x_0 is any solution of $ax/g \equiv 1 \pmod{m/g}$.

- Solving :

  $$\text{For } i = L \text{ to } U \text{ step } s \text{ do } A(i) \sim B(pi + q)$$

  with $f_A(i)$ being $f(i, i, o_A, b_A, n)$ and $f_B(i)$ being $f(i, pi + q, o_B, b_B, n)$.

  In this case $b_A = b_B = 1$. The claim is that the three necessary and sufficient conditions for the equation to hold are : 1) $g = \gcd(p-1, n)$ divides $o_B - o_A - q$ i.e., $(o_B - o_A - q) \equiv 0 \pmod{n}$, 2) the equation is satisfied for $i = L$ and 3) the step of the loop is given by $s = n/g$. In the proof let $s = zn/g + w$ with z, w being integers and $0 \leq w < n/g$. Now

  $$L \equiv \frac{b}{g} i_o + t\frac{n}{g} \pmod{n} \text{ for some } t$$

  Hence

  $$L + s \equiv \frac{b}{g} i_o + (t + z)\frac{n}{g} + w \pmod{n}$$

  If $L + s$ satisfies the constraint imposed by the loop then

  $$L \equiv \frac{b}{g} i_o + (t + \gamma)\frac{n}{g} \pmod{n}$$

  or $(\gamma - z)n/g \equiv w \pmod{n}$ or $gcd(n/g, n)|w$ or $(n/g|w)$ or $w = 0$ since $w < n/g$. Hence proved.

- Solving

  $$\text{For } i = L \text{ to } U \text{ step } s \text{ do } A(i) \sim A(i + F)$$

  with $f_A(i)$ being $f(i, i, o, b, n)$.

  For solution we can say the following :

  1. If $F \equiv 0 \pmod{nb}$ then the constraint is satisfied forall i. For proof note that under the assumption on F,

     $$\lfloor \frac{i - o + F}{b} \rfloor \% n = \lfloor \frac{i - o + \gamma nb}{b} \rfloor \% n$$

     which is equal to $f_A(i)$ for all i.

  2. If F is not a multiple of nb then IRANGE (the range of i from $L \ldots U$) cannot cross any processor boundary, and vice-versa.

     If IRANGE crosses a processor boundary then $F \equiv 0 \pmod{nb}$ is a necessary and sufficient condition. For proof assume that $F = \delta nb + \mu, 0 \leq \mu < nb$. Then

     $$\lfloor \frac{i - o + \mu}{b} \rfloor \pmod{n} = \lfloor \frac{i - o}{b} \rfloor \pmod{n} \; \forall i \in \{L \ldots U\} \qquad (5.2)$$

assume that $i = zb + (o\%b)$ satisfies 5.2. Then

$$z + \lfloor \mu/b \rfloor \equiv z \pmod{n}$$

or

$$\lfloor \mu/b \rfloor \equiv 0 \pmod{n}$$

since $\mu < nb, \mu/b < n$, hence $0 \leq \mu < b$ Now if IRANGE crosses a processor boundary then $i = zb + (o\%b) - 1$ also satisfies 5.2. Thus $\mu = 0$.

If IRANGE does not cross a processor boundary i.e., $U - L < b$ and $f_A(L) = f_A(U)$ then (i) if $F \equiv 0 \pmod{nb}$ say that there is no communication otherwise verify for $i = L$ and $i = U$ to determine communication.

3. Solving

$$\text{For } i = L \text{ to } U \text{ step } s \text{ do } A(i) \sim B(i + F)$$

with $f_A(i)$ being $f(i, i, o_A, b, n)$ and $f_B(i)$ being $f(i, i + F, o_B, b, n)$.

For solution note that we can reduce to the previous case by changing F to $F + o_B - o_A$.

4. Solving

$$\text{For } i = L \text{ to } U \text{ step } s\text{do } A(i) \sim B(i + F)$$

with $f_A(i)$ being $f(i, i, 1, b1, n1)$ and $f_B(i)$ being $f(i, i + F, 1, b2, n2)$. Let $R = U - L + 1$ Let $z = \lfloor \frac{R-1}{b1} \rfloor = \lfloor \frac{U-L}{b1} \rfloor$.

Assume the notation [I] to mean that $i = I$ satisfies the relevant communication determining equation and $\neg[I]$ to mean that it does not . Also let FALSE and TRUE denote that relevant equation has been determined to be satisfied and satisfied respectively. Then we consider the following subcases:

(a) $z = 0$
   (1) $\neg[L] \Rightarrow$ FALSE
   (2) $\neg[U] \Rightarrow$FALSE
   (3) $b2 < R \Rightarrow$FALSE
   (4) TRUE

(b) $z = 1$
   (1) $n2 = 1 \Rightarrow$ FALSE
   (2) $\neg[i^*] \Rightarrow$ FALSE where i* is the processor boundary index, i.e.

the value of the loop variable i which is the last element belonging to a processor in this block, in other words, $i^* + 1$ is with the next processor.
(3) $\neg[i^* + 1] \Rightarrow$FALSE
(4) $b2 \leq max(i^* - L, U - i^* - 1) \Rightarrow$ FALSE
(5) TRUE

5.3 Overall strategy of distribution determination

When the user makes some modifications to sequential code, the unknown distributions of newly introduced array variables are determined by :

1. Matching the corresponding formal and actual parameters in the case of a subroutine/function call. In this case the distribution is found out when interprocedural compilation is being carried out. (see next chapter for the various phases of interprocedural compilation)

2. Matching the corresponding array variables in different subroutines declared under the same block. This case is also handled in a interprocedural compilation phase.

3. Array alignment–carried out by us as detailed above.

Chapter 6

Theoretical Issues in Code Generation

The third phase of our system, *the code generator* utilizes a code generation strategy based on the *owner computes* rule – where each processor only computes values of data it owns. Fortran D data decomposition specifications are translated into mathematical distribution functions that determine the ownership of local data. By composing these with subscript functions or their inverse, the compiler can partition the computation and determine nonlocal accesses at compile-time (It may be recalled that our system does not handle the generation of new communications). This information is used to generate efficient SPMD program for execution on the nodes of the distributed-memory machine. The issues mentioned in this chapter are discussed in detail in [9]

6.1 Compilation Example

We compare two different approaches for compiling Fortran programs.

6.1.1 Run-time resolution

A simple compilation technique known as *run-time resolution* yields code that explicitly calculates the ownership and communication for each reference at run-time. Run-time resolution does not require much compiler analysis, but the resulting programs are likely to be extremely inefficient. In fact, they may execute much slower than the original sequential code.

There are several reasons for the poor performance of run-time resolution. First, parallelism is mostly lost because each processor must execute the entire program. Worse still, not only does the program have to explicitly check every variable reference, it has to generate a message for each nonlocal access.

6.1.2 Compile-time resolution

In comparison, when extensive compile-time analysis is performed, the compiler can produce highly efficient code by using a combination of reducing loop bounds and guarding individual statements. Computations are partitioned efficiently to exploit parallelism.

6.2 Formal Model

In this section we provide formal description of the compilation model. Terminology and notation used are described. We begin by examining the algorithm used to compute a simple loop nest using the owner computes rule. Correct application of the rule requires knowledge of the data decomposition for a program which is provided by ALIGN and DISTRIBUTE statements of Fortran D language.

6.2.1 Distribution functions

Data distribution functions specify the mapping of data arrays. The distribution function μ, defined below [9],

$$\mu_A(\vec{i}) = (\delta_A(\vec{i}), \alpha_A(\vec{i})) = (p, \vec{j})$$

is a mapping of the global index $\vec{i}$ of the array A to a local index $\vec{j}$ for a unique processor p. Each distribution function has two component functions δ and α. These functions are used to compute ownership and location information. For a given array A, the owner function δ_A maps the global index $\vec{i}$ to its unique processor owner p, and the local index function α_A maps the global index $\vec{i}$ to a local index $\vec{j}$.

6.2.2 Regular distributions

The formalism described for distribution functions are applicable for both regular and irregular distributions. An advantage of the simple regular distributions is that their corresponding distribution functions can be easily derived at compile-time. For instance, given the following regular distributions,

DIMENSION SEV(N),AP(N,N)
DISTRIBUTE SEV(BLOCK(P))
DISTRIBUTE AP(BLOCK(P),:)

the distribution functions are :

$$\mu_{SEV}^{(block)}(i) = (\lceil i/BlockSize \rceil, (i-1) \bmod BlockSize + 1)$$

$$\mu_{AP}^{(block,:)}(i,j) = (\lceil i/BlockSize \rceil, ((i-1) \bmod BlockSize + 1, j))$$

where P indicates the number of processors, and

$$BlockSize = \lceil N/P \rceil$$

6.2.3 Computation

Computation is represented by the following simple loop nest:

DO $\vec{i} = \vec{l}$ to $\vec{m}$ by $\vec{s}$
    $X(g(\vec{i})) = Y(h(\vec{i}))$
enddo

In this loop-nest $\vec{i}$ is the set of loop iterations. It is also displayed as the triplet $[\vec{l} : \vec{m} : \vec{s}]$. In addition, X and Y are distributed arrays, and g and h are the subscript functions for the left hand side and right hand side array references, respectively.

6.2.4 Image, local index sets

The *image* of an array X on a processor p is defined as follows [9]:

$$image_X(p) = \{\vec{i} | \delta_X(\vec{i}) = p\}$$

which is the set of all array indices that cause a reference to a local element of array X, as determined by the distribution functions for that array. Hence, *image* describes all the elements of array X assigned to a particular processor p. This processor is denoted by t_p.

6.2.5 Iteration sets

The *iteration set* of a reference R for a processor p is defined to be the set of loop iterations $\vec{j}$ that cause R to access data owned by p. The iteration set can be constructed in a very simple manner. The iteration set for a processor p with respect to reference $X(g(\vec{k}))$ is simply $g^{-1}(image_X(p))$.

6.3 Parallel Code Generation

Once the parallelism in the sequential program has been expressed in terms of the distribution functions, we need to use this information and apply the owner's compute rule to cut the loops operating on the partitioned data and find out where

communication needs to be introduced for non-local data accesses. This can be performed in two ways with differing levels of sophistication. For loop-nests which can be characterized at compile-time and have a relatively straightforward loop bound structure (e.g. constant bounds), we can use compile-time resolution to parallelize the code.

6.3.1 Run-time resolution

When the loop-nests cannot be characterized effectively at compile-time (e.g. because of non constant loop bounds, complicated subscript functions etc.), we need to resort to run-time resolution. In this approach, the loops are not cut but the data is cut. If the input program is :

do $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$
  $X(g(\vec{k})) = Y(h(\vec{k}))$
enddo

The parallel run-time code is:

do $\vec{k} = \vec{l}$ to $\vec{m}$ by $\vec{s}$
  if myproc $= owner(Y(h(\vec{k})))$ then
    $sendY(h(\vec{k}))toowner(X(g(\vec{k})))$
  if myproc $= owner(X(g(\vec{k})))$ then
    $recvY(h(\vec{k}))fromowner(Y(h(\vec{k})))$
    $X(g(\vec{k})) = Y(h(\vec{k}))$
  endif
enddo

The feasibility of run-time resolution is based on the assumption that evaluation of guards is much less expensive than a floating point operation. If this is true, run-time resolution is a relatively straightforward parallelization strategy for those loop-nests which cannot be handled by compile-time resolution effectively (this would include the vast majority of loop-nests in typical real code). It may be be noted

that compile-time resolution is not always possible and in such cases it is necessary to use run-time resolution. We however do not generate new communications (the above example has just been mentioned to illustrate the run-time resolution strategy.)

6.3.2 Compile-time resolution

When the loop-nests can be characterized effectively at compile-time, efficient code can be generated at compile-time. This may be accomplished by a combination of reducing loop bounds and guarding individual statements. Both algorithms are presented below [9]

Loop bounds reduction

The algorithm for loop bounds reduction is given below. The algorithm works as follows. First, the iteration sets for all the lhs are computed for the local processor t_p. The loop bounds are then set to the union of all these sets.

for each loop nest $\vec{k} = [\vec{l} : \vec{m} : \vec{s}]$ do
  ReducedIterSet = ϕ
  for each $statement_i$ in loop with lhs = $X_i(g_i(\vec{k}))$ do
    IterSet = $g_i^{-1}(image_X(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
    ReducedIterSet = ReducedIterSet $\cup$ IterSet
  endfor
  reduce bounds of loop nest to those in ReducedIterSet
endfor

Guard introduction

In the case when all assignment statements in a loop nest have the same iteration sets, we do not need any individual statement masks. However, if this is not true,

we need to introduce guards for individual statements. The algorithm given below also clubs together successive statements having the same mask.

for each loop nest $\vec{k} = [\vec{l} : \vec{m} : \vec{s}]$ do
  PreviousIterSet = $[\vec{l} : \vec{m} : \vec{s}]$
  for each $statement_i$ in loop in order do
    if $statement_i$ = assignment and lhs = global array $X_i(g_i(\vec{k}))$ then
      IterSet = $g_i^{-1}(image_X(t_p)) \cap [\vec{l} : \vec{m} : \vec{s}]$
    else
      IterSet = $[\vec{l} : \vec{m} : \vec{s}]$
    endif
    if IterSet = PreviousIterSet then
      insert $statement_i$ after $statement_{i-1}$
    else
      terminate previous mask if it exists
      create new mask for IterSet and insert $statement_i$ inside mask
      PreviousIterSet = IterSet
    endif
  endfor
endfor

6.4 Interprocedural Compilation

In this section we describe the phases of interprocedural compilation performed by our system. We first build the call graph of the various routines in our project. This call graph is used in the *Reaching Decompositions* procedure described next.

6.4.1 Reaching decompositions

Interprocedural compilation is necessary to know the data decomposition of a variable at every point it is referenced in the program. Procedures inherit the data decompositions of their callers. For each call to a procedure, formal parameters inherit the decompositions of the corresponding actual parameters passed at the call, and global variables (passed through the COMMON statement) retain their decomposition from the caller. When a user adds a new array variable to the sequential code, it may be that its decomposition is provided only by matching the corresponding actual and formal parameters of a particular subroutine. So the problem is really circular. Not only do procedures inherit decompositions from their callers, but distributions of arrays in the callers might also be determined only by reverse inheritance from the corresponding formal parameter. We break this circularity by making two passes of the reaching decompositions procedure (detailed below) over the call graph. One a top-down pass and another in the reverse direction. In both these passes the distributions are also determined for the corresponding COMMON variables in all the procedures in the project. During local analysis we calculate the decompositions that reach each call site C. Formally (see [9],

LOCALREACHING(X) = $\{< D, V > | D$ is the set of decomposition specifications reaching actual parameter or global variable V at point X$\}$.
LOCALREACHING may include elements of the form $< T, V >$ if V may be reached by a decomposition inherited from a caller. Formally,

REACHING(P) = $\{< D, V > | D$ is the set of decomposition specifications reaching formal parameter or global variable V at procedure P$\}$.

Reaching Decompositions Algorithm [9]

{* *Local analysis phase* *}
for each procedure P **do**
    initialize decomposition of all variables to T
    for each call site C in P **do**
        calculate LOCALREACHING(C)
    endfor
endfor
{* *Interprocedural propagation phase* *}
for each procedure P **do** (in topological order)
    calculate REACHING(P) =
        $\cup_{P \text{invoked at } C} Translate$(LOCALREACHING(C))
    clone P if multiple decompositions found
    for each call site C in P **do**
        for each element $< T, X > \in$ LOCALREACHING(C) **do**
          replace with $< D, X > \in$ REACHING(P)
        endfor
    endfor
endfor
{* *Interprocedural code generation phase* *}
for each procedure P **do** (in reverse topological order)
    calculate LOCALREACHING for all variables in P
endfor

The function *Translate* maps actual parameters in the LOCALREACHING set of a call to formal parameters in the called procedure. Global variables are simply copied,and actual parameters are replaced by the corresponding formal parameters. (Another issue is handling of *array reshaping* of array variables across the procedure boundaries that we have also taken care of, albeit in the restricted sense.) REACHING(P) is computed as the union of the translated LOCALREACHING sets for all calls to P. We then update all LOCALREACHING sets in P that contain T. Each element $< T, V >$ is expanded to $< D, V >$, where D is the set of decompositions for variable V in REACHING(P). This step propagates decompositions along paths in the call graph. During code generation the compiler needs to determine

which decomposition reaches each variable reference. It repeats the calculation of LOCALREACHING for each procedure, taking REACHING into account.

6.4.2 Array reshaping

Another involved issue with interprocedural compilation is of array reshaping across procedure boundaries. We have also handled this issue, although to a limited extent. A simple case in which distributions of reshaped arrays may be mapped is when a dimension has been collapsed to give a larger array. If the distribution of the collapsed dimension is *replicated* or *sequentialized* then the distributions of the reshaped array can be found out by simply multiplying the block size of the latter array by the size of the collapsed dimension (in the latter case). Our system also handles some other cases of array rehsaping in which a dimension may be exploded to yield more dimensions.

6.5 Discussion

We have shown how the compiler utilizes compile-time analysis to avoid the inefficiencies of run-time resolution. Its code generation strategy is based on the owner computes rule. Fortran D data decomposition specifications are translated into mathematical functions that determine the ownership of local data. By composing these with subscript functions or their inverse, the compiler can partition the computation and determine nonlocal access at compile-time. This information, along with that obtained by various phases of interprocedural compilation, helps the third component of our system (the code generator) to generate efficient compile-time parallel programs.

Chapter 7

Conclusion and Directions for Future Work

The central idea of this thesis is that automating the process of making modifications to sequential code into a given parallelization strategy is more feasible and desirable than attempting completely automatic parallelization. Our tool demonstrates that with minimal language and run-time support, efficient distributed memory MIMD programs can be produced with little effort. In this chapter we summarize the work embodied in this thesis. We present the status of the tool as of now and consider areas for future work.

7.1 Summary and Current Status

Our prototype tool for automatically incorporating modifications to sequential code is at a fairly mature stage. The program analyzer and restructurer (the first component of our system) is a sophisticated tool in itself that is capable of 1) understanding the user annotations and utilizing them purposefully for code generation, 2) Building the annotated parse tree of a sequential fortran program and doing a first pass analysis of the program. The second component that determines the unknown distributions has also been implemented and tested on a large number of programs. The third component of our system, the compile-time code generator is an advanced parallel compiler that performs interprocedural compilation and can generate efficient compile-time code for the *real* ,for example, the weather code. Almost all

subroutines of the routine *PGloopa.f* (those subroutines that do not involve any communication) have been parallelized (using the parser for Fortran D) and tested to yield correct results. We have been able to implement Compile-Time resolution for complicated loop nests and subscript functions, including the complicated access patterns of the linearized arrays.

7.2 Future Work

There are many ideas which can be realised on the basic platform provided by this tool.

7.2.1 Removal of user annotations

Our system requires the user to guide it in the process of parallelization by providing it with some directions in the form of *user annotations* described in Chapter 2. Removal of the necessity of user providing these guidelines would be very desirable, even though, as also mentioned previously, it would place considerable burden on the system.

7.2.2 Removal of communication free restraint

Our system only considers programs that do not involve any communication. The system needs to be extended to those cases when there are communications in the parallel code. In such cases, different alignment techniques will have to be used and conflicting constraints will have to be weighted according to the quality measure of the amount of communication generated by it. Another issue that would be needed to be handled then will be the optimization of communications, the most important factor when communications are introduced in a parallel program.

7.2.3 Optimizations

The SPMD program being generated at present is fairly efficient. However, there is scope of providing more optimizations. In our system we do perform some prelim-

inary optimizations like taking the union of guards of consecutive statements and dumping the unioned guard, rather than dumping the same guard for every consecutive statement. Possibility of optimization in terms of dumping of guards needs to further investigated in case of *FOR* loops and procedure/subroutine calls.

7.2.4 Automatic data partitioning

The goal of an automatic data partitioner is to choose an efficient data decomposition. Several researchers [2], [3], [4], [5], [12], [6] have focussed on this aspect, but lot of work is still needed to be done.

7.2.5 Automatic parallelization

Ofcourse the most ideal thing would be to be able to develop a fully automatic parallelizing tool that parallelizes any sequential code without any user intervention. But till the time this objective is fulfilled, researchers will have to come up with less ambitious systems. Our thesis can be considered to be a step in that direction.

Appendix A

Tutorial Introduction to Our System

We have also built a *Graphical User Interface* for ease in using our system. The system can be invoked by the command *xdemo* in the directory (at *dhoop*)

/others/nmittal/fparse/sage/sigma/code/xcode

The file BT_README in the same directory also gives some instructions in handling our system. When one invokes *xdemo*, a screen similar to that shown in figure A.1 (the figure shows the screen when the *File* menu button has been selected) appears on the screen. The menu options are explained as under:

- *File* Using this menu option, one can load, save and edit one's files as in a text editor. The editor supports most of the commands of the *emacs* editor for insertion, deletion and other editing functions. The scrollbar or the arrow keys can be used for scrolling the window.
- *Options* This menu option allows the user to change fonts so that he can see the file in a larger/smaller font than the default font.
- *Parse* Pressing this menu button invokes the first component of our system, that performs program analysis and restructuring. If analysis is completed successfully, then a small message box saying the same appears after some time. Any errors in invocation are put on the window from where *xdemo* was invoked.

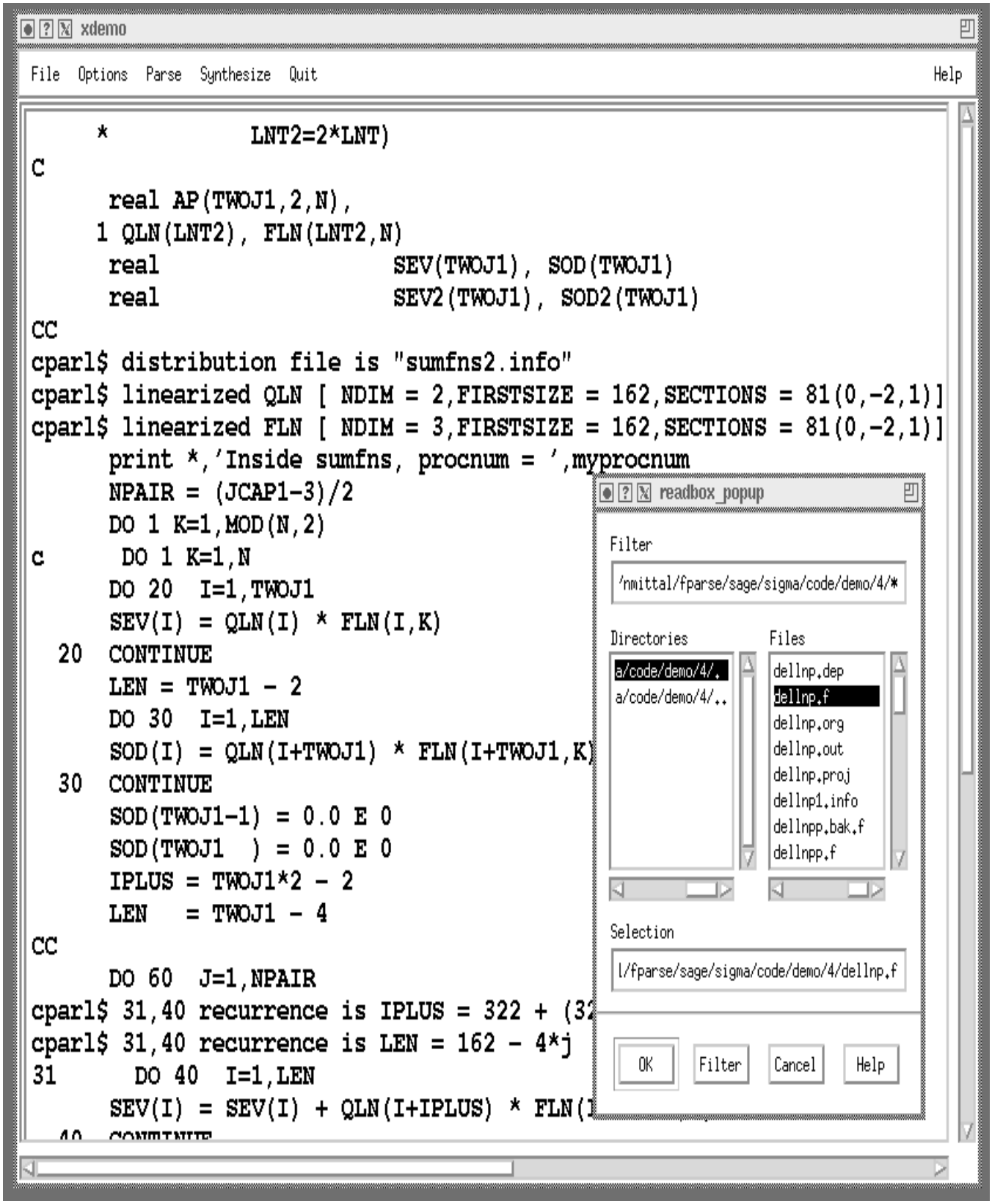

Figure A.1: Sample view of the user interface

- *Synthesize* Pressing this menu button invokes the distribution finder and then the compile-time code generator. After the two functions have been completed successfully, a message box indicating the same appears on the screen. Then the user can view the generated parallel file and run it on *PVM* to find out the results. Any errors/messages in invocation are put on the window from where *xdemo* was invoked.

Bibliography

[1] D. Atapattu, *Performance prediction of supercomputing programs*, PhD Thesis, Indiana University, Bloomington, Indiana,1991.

[2] V.Balasundaram, *A mechanism for keeping useful internal information in paralell programming tools: the data access descriptor*, Journal of Parallel and Distrib. Comput.", 9,154-170(1990)

[3] V.Balasundaram, *Interactive parallelization of numerical scientific programs.* PhD thesis, Department of Computer Science, Rice University, May 1989.

[4] V.Balasundaram et al., *J.PTOOL: A system for static analysis of parallelism in programs*, Tech. Rep., TR88-71, Department of Computer Science, Rice University, June 1988.

[5] V.Balasundaram et al., *The Parascope editor: An interactive parallel programming tool.*, Supercomputing 89,Reno, Nevada, Nov.1989

[6] S.Chatterjee et al., *Automatic-array alignment in data-parallel programs*,RIACS Technical Report 92.18,October 1992.

[7] G. Banga and G. Hasteer, *Semi-automatic parallelization of sequential code*, B.Tech Thesis, 1994.

[8] V.A. Guarna, D.Gannon et al., *An environment for programming parallel scientific applications*, Supercomputing 88, Nov.1988.

[9] Chau-Wen Tseng, *An optimizing fortran D compiler for MIMD distributed-memory machines*, PhD Thesis, Rice University, Houston, Texas, 1993.

[10] D.Gannon et al., *SIGMA II : A tool kit for building parallelizing compilers and performance analysis systems.*,Indiana University Technical Report,1992

[11] M.Gupta and P.Bannerjee, *Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers*, IEEE Trans. on Parallel and Distributed Systems, March 1992.

[12] Jingkeli and Marina Chen, *The data alignment phase in compiling programs for distributed-memory machines*, IEEE Trans. on Parallel and Distrib. Computing, 13,213-221(1991).

[13] S. Nog and A. Sharma, *Distributed memory parallelization of weather forecasting code*, B.Tech Thesis, 1994.