Studies on Program Visualization Systems for Parallelization

笹倉, 万里子

https://doi.org/10.11501/3166971

出版情報：九州大学, 1999, 博士（工学）, 論文博士
バージョン：
権利関係：

Studies on Program Visualization Systems for Parallelization

Mariko Sasakura

Studies on Program Visualization Systems for Parallelization

Mariko Sasakura

February 2000

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis describes a visualization system for helping users understand the analysis of a program extracted by a parallelizing compiler. This introductory chapter explains the backgrounds of this research and gives the outline of this thesis.

1.1 Background

In this section, we explain the brief history of parallel computing and the importance of parallelizing compilers in it. Then, we make clear the problems on current parallelizing compilers, because this research have been done with parallelizing compilers in the phase of compilation.

1.1.1 Parallel Computing

The history of parallel computing is long. Although processor power has improved, there have always been demands that require more computational power, for example analysis of large amounts of data such as simulations of atmosphere or molecules. Therefore, many researchers have studied parallel computing, which aims to execute a program on multiple processors to shorten the time needed for results.

The first practical outcomes of parallel computing research were vector processing and vectorizing compilers. Vector processing deals with array data (vectors) in parallel by pipelining the processors. It looks like a conveyer belt for data. Currently, many vector processors are in practical use, but the users may not be aware that they are using them, because a vectorizing compiler generates a parallel program from the user's original sequential program automatically and the parallel program is executed on the vector processor.

After the success of vector processing came studies of parallel processing. In this thesis, parallel processing means the execution of a program on multiple processors. The processors are either scalar processors or vector processors.

The difference between scalar processors and vector processors mainly appears in the phase of code generation. Some researchers are claiming the difference should be considered at the phase of coding or compilation, but they do not have any remarkable result yet. Therefore, in this thesis, we think there is no problem to assume all processors are scalar, because our

research has been done in the phase of compilation.

The memory model of parallel processing is classified in shared memory, which can be accessed by all processors with the same cost; distributed memory, which is localized memory accessed by only the owner processor; or mixture of shared and distributed memory. We have to deal with different problems depending on the memory model we choose.

The problems on shared memory model is simpler than other memory models. Since all processors can access to data on a shared memory by the same cost, there is only one problem in executing a program on a shared memory system: how to divide a sequential program into plural parts and distribute them on each processor. On distributed memory model, in addition to distirbution of computation, we also consider how divide data and distribute them on each distributed memories.

In this thesis, we call multi-scalar processors with shared memory a *shared memory multi-processor system.* Many researches have been accomplished on a shared memory multi-processor system, because executing a program on it is simpler than on other types of multi-processor models. This thesis mainly discusses about a shared memory multi-processor system, since our work provides ways of using outcomes of previous research.

The programming methods for a shared memory multi-processor system are classified in two ways. One is the method for developing a parallel program with a parallel language. The other is to develop a sequential program with a traditional language and parallelize it with a compiler. We call such compilers parallelizing compilers. A parallelizing compiler transforms a sequential program to a parallel program without changing its semantics.

Basically, a parallelizing compiler reconstructs a program without changing algorithms used in the sequential program.

The advantage of developing a program using a parallel language is that the program is well-tuned to its purpose and may have higher parallelism. The disadvantages of developing a program using parallel languages are as follows:

- We must learn a parallel language.
- We must develop a new parallel algorithm to obtain efficient parallelism.
- We cannot apply the knowledge and skills of programming acquired by developing sequential programs.

On the other hand, the advantages of using parallelizing compilers are the following:

- We can use familiar programming languages for development. We don't have to learn new languages.
- We can use familiar algorithms to solve problems. We don't have to develop new algorithms.
- We can apply the knowledge and skills of programming acquired by developing sequential programs. Sometimes we can parallelize a program that is old but sophisticated and bug-free. A parallelizing compiler is useful especially when the original developer of a program is no longer engaged in the maintenance of the program.

The disadvantage of using parallelizing compilers is that a program generated by a parallelizing compiler may have less parallelism than a program developed in a parallel language.

Usually, users who need parallel computing are experts in physics, chemistry, atmosphere, not in computer science. Since it takes great effort for them to learn new languages and develop parallel programs from scratch, parallelizing compilers can help them. Parallelizing compilers can also parallelize old programs without modifications. Therefore, parallelizing compilers are an essential tool for parallel computing.

1.1.2 Problems on parallelizing compilers

As we mentioned in the previous section, parallelizing compilers are an essential tool for novice users. Although parallelizing compilers try to transform any sequential program into a parallel program automatically, they often fail to extract some kinds of parallelism from sequential programs. Of course, a parallelizing compilers may extract all kinds of parallelism if it does precise analysis of a program with high cost. However, sometimes the cost is too high for users, therefore, users prefer giving up the preciseness. Then, compilers may miss some kinds of parallelism. In other words, there is a trade-off between preciseness and cost of analysis.

Usually, a parallelizing compiler parallelizes only loops in a program, because loops are considered to have higher potential parallelism than other parts. To generate a parallel program from a sequential program, a parallelizing compiler takes three steps:

1. It analyzes control flows in a given program.

2. It analyzes data dependences in loops.

3. It reconstructs loops that can be parallelized.

We can use an established method for the analysis of control flow [1], but we still have problems in the analysis of data dependence.

The analysis of data dependence on the source level checks when each variable and array is accessed. The cost of analysis, which is determined by how much time it takes, depends on the preciseness of the analysis. Two examples are listed below:

- The preciseness of the analysis about indixes of arrays. It is expensive to analyze complex expressions such as more than linear expressions, or indirect accesses that occur when there is an array in the index.

- The preciseness of the analysis of data dependence of inter procedures. That is more expensive than the analysis of inner procedure.

If we choose the low cost method and omit precise analysis, we can miss some loops that have potential parallelism. But, it is useless to take longer time to analyze a program than to execute the program in sequential processing. So, sometimes users choose the low cost but non-precise analysis.

We also have problems with the reconstruction of loops by a parallelizing compiler. Many reconstruction methods have been developed for many paterns of loops, so that, as the first problem we have to choose a way to reconstruct loops. Since several reconstruction methods may be available,

we must choose only some of them and decide on the proper order to apply them. Sometimes, the efficiency of the methods depends on the execution environment of the program, for example, the number of processors we can use. Therefore, there is no specific answer to which methods we should choose.

The second problem is that we don't know whether a loop can be parallelized and, if it can be, which method brings more parallelism than others unless we have precise analysis of data dependences. Because precise analysis is expensive, we cannot expect precise analysis by a parallelizing compiler. If the compiler finds no data dependence, we can believe that there is in fact no data dependence. But, if the compiler does not find evidence that there is no data dependence, we should assume that there are in fact data dependences to avoid fatal misunderstanding. Consequently, some loops that have no data dependence but are not found by the compiler may not be parallelized.

Currently, users parallelize their sequential programs in the following way. First, they compile the program by a parallelizing compiler and get a parallel program. Then, they execute the parallel program. If they are not satisfied with the efficiency of the parallel program, they rewrite their original sequential program or find a better way to parallelize the program and then repeat the process. However, it is very difficult for users to rewrite programs which have more potential for parallelization because the only information they have is that performance is not very good. There are two problems with this process:

1. Users cannot know which statement in the source code influences which part of the executed code.

2. Users can know the performance of a program only after the execution of the program.

Source-to-source parallelizing compilers may be useful for the second problem. They output parallelized programs in a readable and editable format. Therefore, users can check and rewrite their programs after compilation but before execution. However, source-to-source compilers cannot solve the first problem. By using source-to-source compilers, we can investigate the correspondence between the original program and the parallelized program. But it is still hard for users to check statements in the source code in text format in order to rewrite the program or find a better method of parallelization. This is true for three reasons:

1. Programs are usually large.

2. Users must check the output of the parallelizing compiler, which is often totally reconstructed.

3. Users often need information extracted from source code to rewrite the program or find a better method of parallelization. Two examples of such information are control dependence and data dependence. But this information is also difficult to understand in text format, mainly because there is typically a large amount of data.

To solve these problems, users can show the following information, which is used in a parallelizing compiler:

- The correspondence between source code and execution code.

- Data dependence, which is found by the compiler or trace data in execution.

This information allows users to do the following:

- Give "no data dependence" information to the compilers, if users know there is no data dependence in a loop but the compilers cannot find that.
- Select appropriate methods to reconstruct loops with the information about data dependence given by the compilers and knowledge about the program or algorithms users originally have. They can then indicate them to the compilers.

1.2 Overview of Our Research

In this section, we propose an interactive compilation environment, and present the importance of it. Then we show the structure of this thesis briefly.

1.2.1 Visualization for parallelization

We propose an interactive compilation environment, which means that users and compilers collaborate in compilation. Figure 1.1 shows a concept of this environment.

An interactive compilation environment supports a cycle of compilation and modification of a program. The cycle is almost the same as the current parallelization cycle mentioned above, but it is supported systematically.

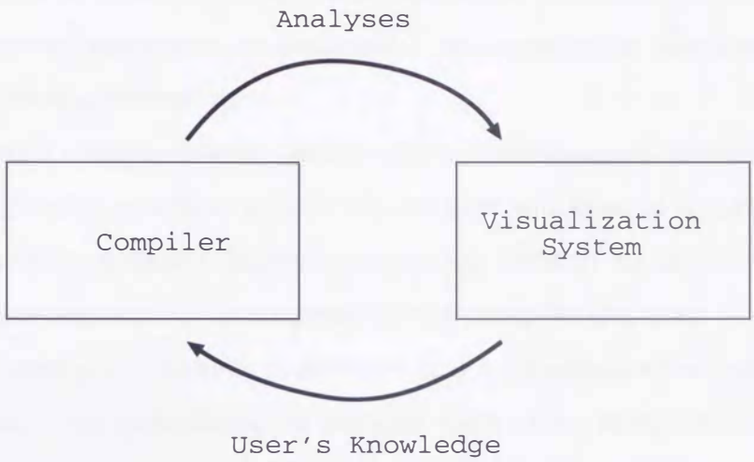


Figure 1.1: An interactive compilation environment

The support provided by a system is divided into two phases. The first is the presentation phase, which informs users about the analysis of a program extracted by a compiler. The second is the feedback phase, which informs a compiler about a user's knowledge and choices.

Several interactive compilation environments have been proposed. We overview these systems in section 2.1.2.

The usefulness of interactive compilation environments is discussed in reference [8]. The authors of this report compared the costs to develop a parallel program and the effectiveness of that program when it is developed with a parallel language, generating a parallel program from a sequential program automatically by a parallelizing compiler, and developing a parallel program from a sequential program by parallel compilers and users. In the

environment of SGI Origin 2000, the authors found CAPTools [12] (a delegate of interactive compilation environments) developed most effective parallel program by the shortest time.

Although existing interactive compilation environments are useful, they are not powerful enough to support novice users who need to cooperate with a parallelizing compiler. The most important point of an interactive compilation environment is the cooperation of a compiler and users. Therefore, the most important problem in implementing an interactive compilation environment is the communication between users and a compiler. There are two specific problems:

1. How a compiler provides analyses of a program to users.

2. How users inform the compiler about their knowledge and decisions.

In the research presented here, we focus on the communication from a compiler to users. Existing systems represent analyses by a compiler only in text form or simple two-dimensional graph representation. For example, one of the most famous interactive compilation systems, ParaScope Editor[10], [14], has no graphical function to show users the analysis by a parallelizing compiler. CAPTools only shows draw control and data dependence graphs as two-dimensional graphs. SUIF[18], [30] uses three-dimensional graphics to visualize call graphs, but that's all.

We visualize the analysis by a parallelizing compilers from a new point of view. We reconstruct traditional representations of the analysis by a parallelizing compiler and make new "views" in three-dimensional. We think

visualization is one of the methods that helps users understand complex information easily.

Here, we propose a visualization system called NaraView, which visualizes the information from parallelizing compilers to help users understand the reconstructed programs and the control and data dependence of the programs. The main idea behind NaraView is to present detailed information obtained by parallelizing compilers to users visually and, hence, intuitively.

Although NaraView has four views, each of which visualizes a program from a different point of view, in this thesis, we focus on only two of them: the Program Structure View and the Data Dependence View. The Program Structure View visualizes the program structure of a given program with three-dimensional axes representing the program flow, the loop and function call hierarchy, and parallelism. We define *program structure* in section 4.2. The Data Dependence View is a relational map of a given loop, its variables, or the array elements accessed, which may have data dependences in that loop.

1.2.2 Outline of this thesis

This thesis consists of seven chapters.

Chapter 2 presents the background needed to understand our work. In the first section of the chapter, we explain popular loop reconstruction methods to use them later. And also we discuss related works on interactive compilation environments in detail. The second section shows program visualization systems for other fields. We intend to make clear "what is visualization" in

the sectoin.

Chapter 3 gives an overview of NaraView. We make clear its requirements and show basic architecture of NaraView.

In chapter 4, we present the Program Structure View which visualizes a structure of a program. We introduce the inner representation of the Program Structure View, HTGv, and show how we map an HTGv to three-dimensional graphics.

Chapter 5 shows the Data Dependence View which visualizes data dependences in a loop. We propose variable-oriented data dependence model which is the inner representation of the Data Dependence View and explain how we project it in three-dimensional graphics.

Chapter 6 presents examples that show the usefulness of NaraView. We show how we find a loop to be checked in the Program Structure View and how we "read" data dependences from the Data Dependence View.

In chapter 7, we summarize this thesis and discuss further problems.

Chapter 2

Related Researches

This chapter provides an overview of the background of our research on parallelization and visualization. Popular loop reconstruction methods are introduced briefly and the names of the methods are then used without explanation after this chapter. We use the loop reconstruction methods in examples (chapter 6). Then, we summarize the current status of existing interactive compilation environments. NaraView is one of interactive compilation environments, so we make clear differences of it from the previous works. In the last section, we mention our opinion on "what is visualization." And we survey the current research about visualizing programs for various purposes. Since NaraView is also one of program visualization systems, the research mentiond in the section has an influence on NaraView.

2.1 Parallelization

This section surveys two topics about parallelization. Section 2.1.1 presents basic loop transformation methods to give a basic image of loop parallelization. The methods are used in examples in chapter 6. Section 2.1.2 describes existing interactive compilation environments to explain the position of NaraView among them.

2.1.1 Loop reconstruction methods

This section outlines the basic ways in which a compiler parallelizes programs.

Programs are parallelized in various ways with various granularity. For example, fine-grain parallelization for superscalar or very large instruction word(VLIW) systems which have hardware that can execute some machine operations at the same time, is an arrangement of instruction codes for supplying these codes in parallel. Coarse-grain parallelization for PVM or MPI which are libraries for a message-passing model, is usually the partitioning of a program into several parts based on functions and assigning the codes to each processor. On the other hand, parallelization at the source level for a shared memory multi-processor system, which is our target, is usually done by parallelizing loops in a program. Because loops take much of the time needed to execute power programs, parallelizing the loops can drastically reduce execution time.

Generally, parallelizing compilers can parallelize the following loops automatically:

- Loops without data dependence
- Loops without conditional branches and
- Loops without function calls

Currently, some compilers can perform interprocedure analysis, so they can parallelize loops with function calls.

Compilers also try to parallelize loops with data dependence by dividing each loop into two parts. One part has data dependence and the other has no data dependence. The compiler parallelizes the part that has no data dependence. The methods based on this idea are called loop reconstruction methods or loop transformation methods, since loops after parallelization are not the same as the original loops even in source code. The loops are rewritten by a compiler.

Below, we introduce some typical loop transformation methods in brief. Most of them are used in examples in chapter 6.

(a) Loop interchange

Loop interchange [32] exchanges the positions of two nested loops but it cannot interchange all loops. To interchange loops, the data dependence after the loop interchange must be the same as the data dependence of the original program.

There must be at least two loops in a perfect loop nest. For example, the loops in Figure 2.1(a) can be interchanged as in (b). But the loops in Figure 2.2 (a) cannot be interchanged because the data dependences in the loops of Figure 2.2 (a) and (b) are different.

```
      do 10 i = 2, n                do 20 j = 2, n
        do 20 j = 2, n                do 10 i = 2, n
        a(i, j) = a(i-1, j-1)         a(i, j) = a(i-1, j-1)
20      continue                10    continue
10    continue                  20  continue
```

(a) (b)

Figure 2.1: Loops that can be interchanged: (a) original loops (b) interchanged loops.

```
      do 10 i = 2, n                do 20 j = 2, n
        do 20 j = 2, n                do 10 i = 2, n
        a(i, j) = a(i-1, j+1)         a(i, j) = a(i-1, j+1)
20      continue                10    continue
10    continue                  20  continue
```

(a) (b)

Figure 2.2: Loops that cannot be interchanged. (a) original loops (b) interchanged loops. The data dependence of (a) and (b) are different, for example, in the access order to a(2, 3).

We apply loop interchange to parallelize loops when only one loop can be parallelized. If the target machine is a vector processor, it is better to bring a loop that can be parallelized into a loop that cannot be. If the loop with the largest range can be parallelized into the innermost position of the loops, vectorization is improved. On the other hand, if the target machine is a multi-processor, it is better to bring a loop that cannot be parallelized into a loop that can be. If the loop with the largest range can be parallelized outward in the loop nest, parallel performance is improved.

(b) Scalar expansion

If variables in a loop are an obstacle to parallelizing, the loop can be parallelized by scalar expansion [32]. Scalar expansion expands a variable into an array. For example, the inner loop in Figure 2.3 (a) can be parallelized, but the outer loop cannot be parallelized because of the data dependence on variable m. However, both loops in Figure 2.3 (b) can be parallelized since scalar expansion on m makes the data dependence disappear.

Scalar expansion can be applied to a variable when the variable is written after read. This is called anti-dependence and is described in section 5.1.1.

(c) Loop distribution

Loop distribution [32] divides a single loop into many. It is used for two tasks:

- To make a perfect loop for applying other loop reconstruction methods
- To divide a loop into two parts. One part has data dependence, the other does not.

Before applying loop distribution to a loop, we should verify whether the

```
      do 10 i = 2, n                    do 20 j = 2, n
        m = b(i) / c(i)                   m(j) = b(i) / c(i)
        do 20 j = 2, n                    do 10 i = 2, n
        a(i, j) = m * a(i, j)             a(i, j) = m(j) * a(i, j)
20      continue                  10      continue
10    continue                    20    continue

          (a)                               (b)
```

Figure 2.3: Scalar expansion. (a) original loops (b) expanded loops.

data dependence allows it. For example, the loop in Figure 2.4 (a) can be distributed as in (b), but the loop in Figure 2.5 cannot be distributed because of the data dependence on array *c*.

(d) Loop skewing

Loop skewing [31] can make wavefront computations in parallel. Wavefront computations are called this because the computations are performed like a wave across the iteration space. Loop skewing reconstructs the iteration space of the wavefront computations.

We can explain loop skewing intuitively by an example. Figure 2.6 (a) is a wavefront computation. This loop has data dependence. Loop skewing reconstructs the loop to look like the loop in Figure 2.6 (b). This loop still has data dependence. But if we apply loop interchange to the loop, the inner loop has no data dependence.

```
      do 10 i = 2, n
        do 20 j = 2, n
        a(i, j) = c(i+1, j+1)
        c(i, j) = b(i, j)
20      continue
10    continue
```

(a)

```
      do 20 j = 2, n
        do 10 i = 2, n
        a(i, j) = c(i+1, j+1)
10      continue
20    continue
      do 40 j = 2, n
        do 30 i = 2, n
        c(i, j) = b(i, j)
30      continue
40    continue
```

(b)

Figure 2.4: Loop distribution (a) original loops. (b) distributed loops.

```
      do 10 i = 2, n
        do 20 j = 2, n
        a(i, j) = c(i-1, j-1)
        c(i, j) = b(i, j)
20      continue
10    continue
```

Figure 2.5: Loops cannot be distributed.

```
      do 10 i = 2, n-1
        do 20 j = 2, m-1
        a(i, j) = a(i-1, j) + a(i+1, j) + a(i, j-1) + a(i, j+1)
20      continue
10    continue
```

(a)

```
      do 20 i = 2, n-1
        do 10 j = i+2, i+m-1
        a(i, j) = a(i-1, j) + a(i+1, j) + a(i, j-1) + a(i, j+1)
10      continue
20    continue
```

(b)

```
      do 20 j = 4, m+n-1
        do 10 i = max(2, j-m+1), min(n-1, j-2)
        a(i, j) = a(i-1, j) + a(i+1, j) + a(i, j-1) + a(i, j+1)
10      continue
20    continue
```

(c)

Figure 2.6: Loop skewing: (a) original loops (b) skewed loops (c) skewed and interchanged loops.

```
      do 10 i = 3, n
        do 20 j = 5, m
          a(i, j) = b(i-3,j-5)
          b(i, j) = a(i-2,j-4)
20      continue
10    continue
```

(a)

```
      do 20 i = 3, n, 2
        do 30 ii = i, i+1
          do 10 j = 5, m
            a(i, j) = b(i-3, j-5)
            b(i, j) = a(i-2, j-4)
10        continue
30      continue
20    continue
```

(b)

Figure 2.7: Cycle shrinking: (a) original loops (b) shrinked loops.

(e) Cycle shrinking

Cycle shrinking [22] reconstructs a loop that has data dependence to double loops, in which the inner loop can be parallelized. For example, the loop in Figure 2.7 (a) has data dependence, so, the outer loop cannot be parallelized. Cycle shrinking turns the outter loop into a double loop as in Figure 2.7 (b). The inner two loops of (b) can be parallelized.

The main idea of cycle shrinking is to make a new loop which has the smaller number of times of execution than dependence distance [3]. Intuitively, dependence distance is the number of times of loop from an access to the next access to the same variable. As the result of cycle shrinking, we get a new loop that can be parallelized, and may be expected more parallelism.

There are other versions of cycle shrinking, such as selective shrinking and TD-shrinking. These versions stem from the same idea, but the concrete ways of applying them differ [22].

Cycle shrinking can be applied to any loop. Sometimes, it improves performance, but sometimes it reduces performance. The change in the performance depends on the interval of data dependence of the loop and the environment in which the loop is executed.

The loop reconstruction methods we introduce here are typical but not all compilers provide all of the methods. Many provide other methods. For more details about loop reconstruction methods, see reference [2].

At the end of this section is an example of how we direct a compiler to apply loop reconstruction methods. The directions differ according to which compiler we use, but the basic idea is the same. So, we explain the case of Parafrase-2. Parafrase-2 is one of the most popular parallelizing compilers in the academic arena. It was developed by Center for Supercomputing Research and Development of Illinois University [23].

Using Parafrase-2, we apply loop reconstruction methods by a pass file (Figure 2.8). In a pass file for Parafrase-2, we direct not only the loop transformation methods but also the analysis methods we want to perform. The loop reconstruction methods written in a pass file are applied to all loops of a program in the order they are written in the pass file. Although it seems wasteful to apply all directed loop reconstruction methods to all loops, an automatic parallelizing compiler has no way to choose which loops to apply to which loop transformation methods.

```
fixup /dev/null        # fixup routines
callgraph /dev/null    # build call graph
flow /dev/null         # flow analysis
donest /dev/null       # determine nesting levels
depend /dev/null       # calculate IN and OUT sets
loop_interchange       # loop interchange
builddep /dev/null     # build dependence graph
dotodoall /dev/null    # find DOALL loops
codegen                # code generation
```

Figure 2.8: Sample pass file for Parafrase-2. The # in each line indicates a comment.

2.1.2 Interactive compilation environments

This section presents an overview of interactive compilation environments. Though some of them are not called so, they have been developed to parallelize a program with the user's input. Therefore, we call them interactive compilation environments.

ParaScope Editor, one of the most well-known interactive compilation systems, was developed at Rice University [10], [14]. ParaScope Editor is a parallelizing compiler with interactive user interfaces. ParaScope Editor provides almost all known loop reconstruction methods at those days. Interactive compilation with ParaScope Editor has three features:

- Users can see data dependence in a program analyzed by ParaScope in

text form.

- Users can inform ParaScope that data dependence does not exist.
- Uers can select a loop from the program and choose a reconstruction method from a menu provided by ParaScope.

The experience of scientific programmers and tool designers who used ParaScope Editor is reported in [10]. Overall, the users appreciated the interactive compilation environment, but they requested improvements in the user interface and analysis.

SUIF is a parallelizing compiler developed at Stanford University [18], [30]. Currently, SUIF allows interprocedural parallelization analysis, dynamic execution analysis, and interprocedural program slicing. SUIF also has a visualization system for call graphs and source code and provides an interactive way to modify a program. A call graph represents a relationship between procedures, namely, which procedure (function or subroutine) calls which procedure. SUIF visualizes it on a sphere. This technique is based on the information in reference [20]. Source code is visualized in SUIF as shrunken text. The text is shrunk to the point that we cannot read its characters but we can still see the outline of it. SUIF colors the shrunken text according to a measure indicated by users, so users can find the important part of the source code, scale up the part, and edit it.

CAPTools, another parallelizing compiler, was developed at Greenwich University [12] and is one of the compilers designed for practical use. Frumkin and colleagues [8] report that CAPTools is useful for interactive parallelization, but its visualization tool is not so powerful. It draws only traditional two

dimensional graphs consisting of circles and edges for control flow graphs and data dependence graphs. But it displays data dependence, which obstructs parallelization of part of a program, in text form. Users can inform the compiler that the data dependence should be ignored, so that the compiler can parallelize the part. This methods may be useful for export users.

Parassist [13] is a parallelization assistance system rather than a parallelizing compiler, although it includes a parallelizing compiler. In addition to static analyses by a compiler, it aims to perform dynamic analyses of a program with the assistance of users. Therefore, it provides window-based interactive user interfaces, but no visualization.

The systems mentioned here collaborate with compilers and present information from the compilers to users in text form or simple two-dimensional figures. NaraView also collaborates with a compiler, Parafrase-2, so basic architecture is same with the systems. However, the main difference of NaraView from the systems is how to present information to users. Since it is difficult for users to understand the meaning of information extracted by compilers, we believe information should be visualized effectively. Therefore, we provide several visualization methods in NaraView.

2.2 Program Visualization

Visualization can be regarded as making pictures of something. To be more precise, we consider visualization to be a process of deleting unnecessary information and choosing only the essential information for a specific purpose. Pictures are the result of visualization. If the visualization is good, the

pictures we get may also be good.

Visualization by the use of computers generates a large amount of data through simulations or the observations of sciences such as fluid mechanics, meteorology, physics, chemistry, and medical science. But scientists have suffered in trying to understand large amounts of data. Therefore, they have been eager to accept visualization. Research about visualization in science is called "scientific visualization". But visualization can deal with other types of data.

Roughly speaking, we can divide visualization into two fields according to what we visualize. In the first, we visualize things which have models. Scientific visualization belongs to this field. The aim of the field is to visualize complex or large amounts of data for easy understanding. In the second field, we visualize things that have no models. The visualized data is not always large or complex. In this field, what we visualize is as important as how we visualize it. In the first field, researchers of visualization can use models proposed by scientists. In the second field, however, researchers must find a suitable model to visualize a target.

Program visualization belongs to the second field of visualization, and NaraView is one of the program visualization systems. In this section, we survey program visualization systems in brief and clearly the position of NaraView among other program visualization systems.

The aim of program visualization is to help users understand a program, which is represented by source code. Users may be able to understand the program by reading the source code, but many programs, especially those in practical use, consist of a large amounts of source code. So, it is not always

easy for users to understand a program, even when they write it themselves. Therefore, many researchers have tried to visualize programs.

Users should be able to understand a program for a variety of reasons:

- Debugging
- Performance tuning
- Maintenance
- Program development
- Education

Visualization for debugging helps users find bugs in a program. Visualization for performance tuning assists users in modifying a program for efficient execution. Visualization used for parallelization belongs in this category. Visualization for maintenance helps users understand structure, algorithms, and other aspects about a program to maintain it. Visualization for program development means to provide a visual language to users to help them write a program. Visualization for education aims to help users learn algorithms that are already developed.

Previous research has also focused on what is visualized:

- Source code (source statements)
- Information extracted from source code
  - call graphs

- control flow graphs
- data dependence

- Information extracted during execution
  - trace data
  - algorithms

Below is an overview of program visualization research.

For debugging, much research visualizes trace data, which is a record of an execution. Since debugging parallel programs is difficult, much work has been done on visual debuggers for parallel programs [17], [27]. These debuggers visualize when a variable is accessed or when a message is passed by an execution. A debugger should show the relationship between the data and source code, but other work visualizes only source code. Koike and Aida have tried visualize source code of the Schema program [15], but this visualization has not yet been used in debugging.

With regard to visualization for performance tuning, many works especially target parallel programs. The visualization of the call graphs in SUIF (see section 2.1.2) can be included here. ParaGraph [11] is another well-known visualization tool that visualizes trace data of programs using PVM. In general, systems that visualize trace data for performance tuning do not focus on the relationship between the data and source code. For example, they visualize how many seconds a processor works or how much memory is used.

Visualization for maintenance is mainly discussed in the field of software

engineering. Koike and Chu [16] proposed to use visualization for version maintenance. Chuah and Eick [6] proposed to use visualization for managing software development.

Visualization for program development, in other words visual languages, has been studied for a long time, and many visual languages have been proposed [7], [19], [26]. Browne and colleagues [5] proposed a visual and parallel language. Unfortunately, the languages have not been popular, because some of them have limited ability or special purposes, some of them are too complicated to use, and some of them make too heavy load for current machines to make a practical programs.

For education, the works called algorithm animation are well-known. They visualize a process in execution from trace data. Reference [28] includes the primary work on algorithm animation.

It is very important for program visualization systems to clarify a goal or target of visualization. NaraView has a clear goal which is to let users understand information from compilers by visualization. Currently, SUIF is only system that has the same goal, but it visualizes only call graphs. About visualization methods, NaraView builds original models for its visualization and has originality on using three-dimensional graphics and giving meaning to each axis.

Chapter 3

NaraView

This chapter provides an outline of NaraView. First we clarify the requirements of NaraView, and make a list of functions that NaraView should provide. Then we show the architecture of NaraView. NaraView consists of four views. Each view visualizes different information by different method.

3.1 Requirements

NaraView is used to present an interface to users of a compiler in an interactive compilation environment. NaraView visualizes the information users need to parallelize a program. This information is extracted by a compiler.

To put it simply, NaraView shows information that allows users to find loops that were not parallelized by a compiler but should be. The loops that should be parallelized are as follows:

- Loops in which the running time can be reduced dynamically by par-

allelization

- Loops that compilers cannot parallelize automatically but users may be able to parallelize

Since compilers and users cannot guess the efficiency of a parallelized loop without executing it, NaraView is designed to identify loops that compilers cannot parallelize but users may.

If users try to parallelize a loop that cannot be parallelized automatically, they must know the following:

1. Which loops are not parallelized automatically.
2. Why they are not parallelized.

Two factors may explain why a loop is not parallelized automatically:

1. There are conditional branches in the loop.
2. There is data dependence in the loop.

A compiler reports data dependence under these conditions:

1. The compiler finds concrete data dependence.
2. The compiler finds that there is no data dependence.
3. The compiler does not find evidence that there is no data. dependence

In the third case, the compiler may not find there is no data dependence for two reasons:

1. There are variables whose values are not known in compilation time, for example, indirect accesses to an array or variables whose values are input by users in running time.

2. There are function calls.

NaraView is required to show the information in these two cases to allow users to select loops that may be parallelized with the user's knowledge. For example, if values of variables in a loop are not known in compile time, a compiler reports that there is data dependence, and the loop is not parallelized. But if users know the values assigned to the variables in execution time, they can easily input the values to the compiler. Then, the compiler can analyze the loop again.

The goal of NaraView is to allow users to specify how reconstruct source code in order to be parallelized based on the information extracted by a compiler. Therefore, users must know which part of the source code corresponds to the information. In other words, we want to inform users which part of the source code should be modified if they want to change the result. So, we believe it is essential to clarify the correspondence between the source code and the information and to present correspondent source code to users, if needed.

With regard to organizing visualization, there are two general phases of investigation. The first phase is to grasp an overview of a target. The second is to investigate the details of the target. There are also two phases to interactive parallelization. The first is to grasp an overview of a program; the second is to investigate the details of each loop. Therefore, we must

provide ways to visualize that satisfy the both phases.

Thus, NaraView must have the following features:

- A function to give an overview of a program
- A function to find loops that are not parallelized automatically but can be parallelized easily with input from users
- A function to show users how source code corresponds to the identified loops
- A function to provide users detailed information, such as control flow and data dependence, about the identified loops

3.2 Architecture

Figure 3.1 shows the architecture of NaraView. NaraView collaborates with Parafrase-2 [23], one of the most popular parallelizing compilers in the academic arena, to furnish an interactive system for more conspicuous parallelization. NaraView provides four views to visualize information extracted by Parafrase-2. A view is a layout of the extracted information, such as which parts are chosen to be displayed and how they are placed. Users can understand the characteristics of the information through the appropriate views, and investigate the best strategy for parallelizing the original programs.

The four views are described below:

Program Structure View (PSV) gathers information about control flow and data dependence and visualizes this information as a kind of map

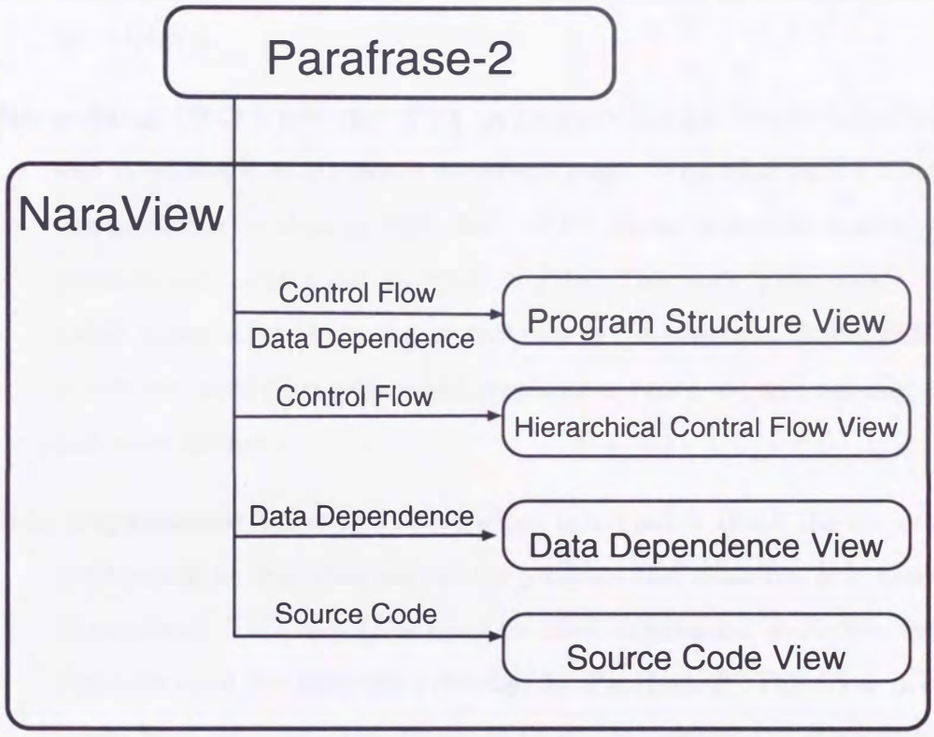


Figure 3.1: The architecture of NaraView

of a given program. This view gives users an intuitive feeling of the program. Therefore, PSV does not show the details of control flow and data dependence but identifies important points and visualizes them. PSV also provides ways to invoke other views that are useful for investigating the program in detail. The details of PSV are discussed in chapter 4.

Hierarchical CFG View (HCFV) gathers information about control flow and visualizes it as a three-dimensional graph. This view uses a mechanism similar to that of PSV, but HCFV shows branches caused by if-statements, which are omitted in PSV. This view gives users detailed information about the control flow of the program. Since HCFV is nothing more than a standard graph viewer tool, we will not discuss this view further.

Data Dependence View (DDV) gathers information about the data dependence of an indicated part of the program and visualizes it in three-dimensional. This view gives users detailed information about the data dependence of the program extracted by Parafrase-2. This view is invoked from the PSV. The details of DDV are discussed in chapter 5.

Source Code View (SCV) shows the source code of an indicated part of the original program. This view is invoked from the PSV. Since this view is a simple editor, we will not discussed it further.

To fulfill the requirements listed earlier in this chapter, PSV takes part in giving an overview to users, and HCFV, DDV and SCV take part in inves-

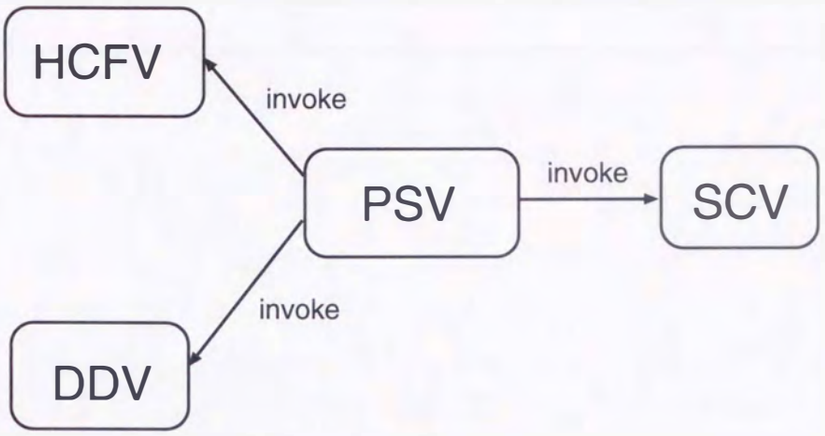


Figure 3.2: The relationship among the views of NaraView

tigating the details of a program. The correspondence between information and the source code are shown by PSV, which connects to SCV to show source code and the other views (Figure 3.2).

Parallelization with NaraView takes place as follows. First, NaraView invokes PSV to give users an abstract and overall impression of the given sequential or partially parallelized program, as well as clues for further investigation. Next, the user specifies a loop to focus on. According to the user's specification, NaraView invokes HCFV/DDV/SCV to show the detail of the control flow graphs and the data dependence in both the loop and in the original source program of the loop. With this information, the user can find unnecessary data dependent parts intuitively and can improve the source program with other parallelization techniques.

Although Parafrase-2 supports both Fortran and C, currently NaraView

supports only Fortran-77, because most users of parallelization still use Fortran-77. But expansion of NaraView to other operative languages is not so difficult.

Chapter 4

Visualization on Program Structures

This chapter describes the Program Structure View of NaraView. First, we introduce Hierarchical Task Graph (HTG), a program representation proposed by Girkar and Polychronopoulus [9]. Then, we propose a new program representation for visualization, HTGv, that is a variation of HTG. In the last section, we explain how we visualize HTGv.

4.1 HTG

Hierarchical Task Graph (HTG) is an intermediate program representation proposed by Girkar and Polychronopoulus [9]. It was developed for Parafrase-2.

Since Parafrase-2 is a multilingual compiler (target languages are For-

tran and C), HTG must be powerful enough to represent both languages for optimization, code generation, and parallelization at all levels of granularity.

HTG is created from a control flow graph (CFG). The importance of a hierarchical structure included in CFG for optimization has been noted [1]. A hierarchical structure is also important to detect and manage parallelism, because parallelization is also performed hierarchically. For example, we must first detect and manage parallelism in each basic block in a given program, then between basic blocks, and between groups of basic blocks, and so on.

Here, a basic block means a sequence of sentences that are executed iff the top sentence of the sequence is executed. In other words, a basic block does not have branches or access points to jump in and out of the block.

HTG is a layered graph that is built based on a strongly connected region of a CFG. A CFG is contained in each layer of an HTG, but it is an acyclic graph unlike the original CFG because a new layer of HTG is created from a cyclic part of the original CFG. Thus it is "built based on a strongly connected region."

A CFG is a directed graph with unique nodes ENTRY and EXIT. ENTRY has no incoming arcs, and EXIT has no outgoing arcs. A sample CFG appears in Figure 4.1 (This example originally appears in [9]). Loops in a CFG can be detected by a depth first search traversal. The result is an HTG. The HTG in Figure 4.2 comes from the CFG in Figure 4.1.

We can then add arcs to correspond to the data dependence in each node in the HTG. The arcs have properties connecting the types of data dependence (flow, anti or output dependence which are discussed in chapter 5) and a variable that causes the data dependence.

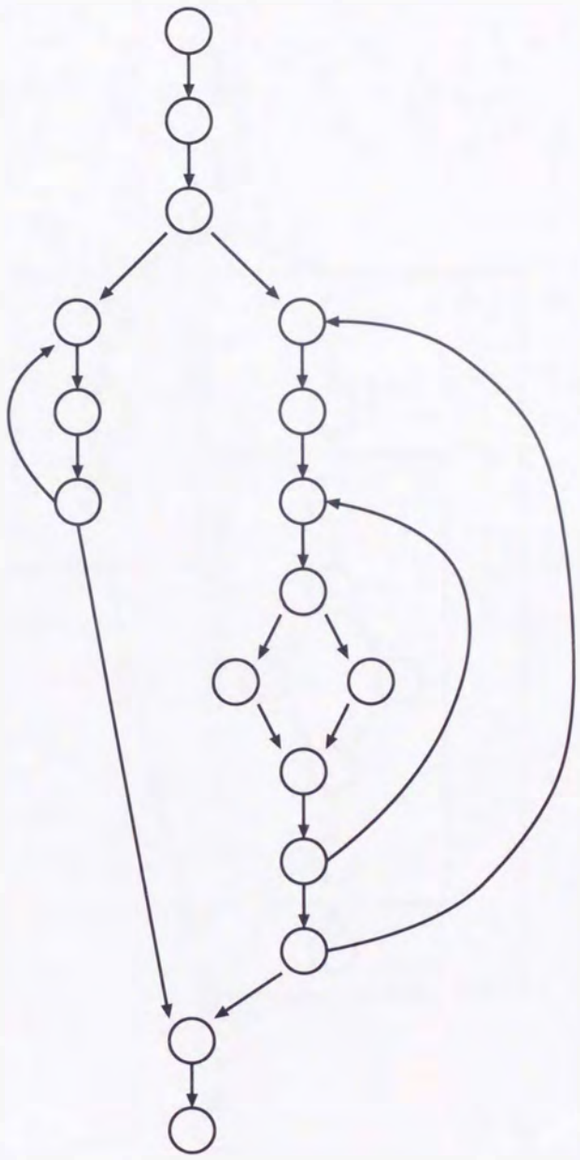


Figure 4.1: A CFG

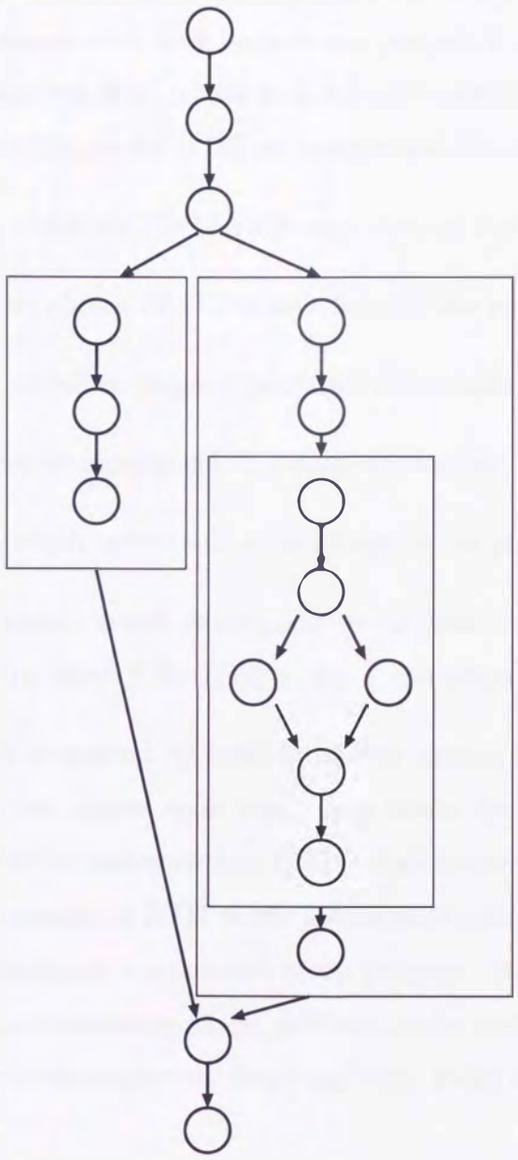


Figure 4.2: An HTG

Although HTG can represent a task graph at all levels of granularity, we discuss HTG at a source code level because our purpose is to visualize at this level. At the source code level, a task in HTG corresponds to a statement of a given program. Nodes in the HTG are categorized into six types:

1. Start nodes, which are ENTRYs in each layer of the graph.
2. Stop nodes, which are EXITs in each layer of the graph.
3. Basic nodes, which correspond to simple statements in the program.
4. Call nodes, which correspond to statements having a subroutine call.
5. Loop nodes, which correspond to the loops in the program.
6. Compound nodes, which correspond to the basic blocks in the flow graph. The top level of the HTG is also a compound node.

General HTGs are generated by making layered graphs from loops in the original CFG. At the source code level, loop nodes are the loops in the program created by DO statements or GOTO statements in Fortran.

The important feature of HTG at the source code level is that each node has a clear correspondence to a sentence of the program. Basic nodes and call nodes have one-to-one correspondence, and loop nodes and compound nodes have one-to-many correspondence. Start and stop nodes do not correspond to statements.

Figure 4.3 shows an HTG format generated by Parafrase-2. The figure shows only two nodes of a program.

```
Node ID: 18
Hierarchy Level: 3
Type: Compound (ID 4  Line 3 to 5 )
Parent: 1
START node: 19
STOP node: 20
HTG arcs from: 24 (FARC) -1
HTG arcs to: 3 (FARC) -1
# of children: 5
Children: 23 22 21 20 19 -1
PDT`PARENT node: 3
PDT`SONS:  -1
CFLongest: 0
CDDDLongest: 0
CDG`IN: arc from 24 to 18
INSET:  -1
OUTSET:  -1
COUNT: 5

  Node ID: 22
  Hierarchy Level: 4
  Type: Basic (Line 4 )
  Code: b[i][j] = a[i - 2][j - 4];
  Parent: 18
  START node: no start node
  STOP node: no stop node
  HTG arcs from: 21 (SEQ) -1
  HTG arcs to: 23 (SEQ) -1
  # of children: 0
  Children: -1
  PDT`PARENT node: 23
  PDT`SONS:  21 -1
  CFLongest: 0
  CDDDLongest: 0
  INSET:  -1
  OUTSET:  -1
  COUNT: 0
```

Figure 4.3: A sample of an HTG format generated by Parafrase-2

4.2 HTG for Visualization

This section describes a variation of HTG for our PSV. The purpose of PSV is to give users a program structure at a glance from the viewpoint of parallelism and to let them select a part of the program to focus on for parallelization.

As program structure is an ambiguous word, we define it from the viewpoint of parallelism. Since almost all parallelization methods at the source level transform loops, we define our program structure through these criteria:

- The number and location of the loops
- The number and location of conditional branches
- The number and location of function calls
- The loops that have already been parallelized

We categorize this information into the following four features:

- A type of nodes, to find a conditional branch or function call
- Program flow, to define the execution order of the nodes
- Hierarchical levels of loop structure, to determine the placement of loops and loop nests
- The measure of parallelism, to show whether a loop has been parallelized

Because HTG is an intermediate representation for a parallelizing compiler, it has information for optimization, code generation, and parallelization. However, some of this information is unnecessary for our purpose,

which is to visualize information to help with parallelization. Therefore, we generate a new graph called *HTGv*, which consists of only the information we need, and we visualize it.

The purpose of visualizing HTGv is to show the program structure to users. The correspondence between HTG and HTGv is as follows:

- Nodes and arcs in HTG are also in HTGv.
- HTG has an acyclic CFG in each layer, but HTGv has an acyclic graph with no branch. If a CFG in each layer of HTG has a branch, we select one path from the CFG and make it a graph of each layer of HTGv that corresponds to the HTG. We cut branches, but we show there are branches by node types mentioned below. Because parallelizing parts of a program that include branches is difficult using the current technique, thus the detail information about branches is not necessary but the information that there is a branch is important for us. In the current version of NaraView, we select the longest path to each branch.
- HTGv has plural nodes that correspond to a node in a parallelized loop in HTG. The plural codes clarify that the loop is parallelized. The number of plural nodes is defined by the number of executions of the loop.

A node in HTGv consists of the following properties:

Node ID. The same number as the corresponding node in HTG.

Node ID for parallelization. The number that distinguishes one node from others that have the same Node ID. In the case of a parallelized loop,

we make plural nodes of HTGv from a node of HTG, so several nodes which have the same Node ID. This number is unique in the nodes that have the same Node ID.

Line number of the source. The line number of the source code that corresponds to this node. It is inherited from the corresponding node in HTG.

Type. Inherited from the corresponding node in HTG. HTG has start, stop, basic, call, loop, and compound nodes, but there are three differences between HTG and HTGv. First, we remove almost all compound nodes by unfolding them because too many hierarchical structures can confuse users. We keep a compound node, which represents the top of the hierarchy of a graph. We call this the root node. Second, we divide the basic nodes into basic, parallel, and if nodes. The information needed for this division is included in HTG. We make the node a parallel node, if it has no data dependence in the basic block. We make the node an if node, if it has more than one outgoing arc in the original HTG. Third, we omit all start and stop nodes because they do not correspond to source code, and we can easily determine where a graph starts and stops without start and stop nodes in visualizaton.

Hierarchical level. Inherited from the corresponding node in HTG.

Hierarchical children. A list of pairs of Node IDs and Node IDs for parallelism. It is a list of children in the HTGv.

Sequence number of execution. HTGv has no branch, so we can number nodes in the order of execution. Root node and loop nodes are unique because they represent a set of nodes. Therefore, we define the sequence number of the execution of a root node or a loop node as the mean of sequence numbers of the execution of its children.

4.3 Program Structure View

This section describes the Program Structure View (PSV), which is a visualization of HTGv. In PSV, we visualize a node as a colored cube. When we visualize HTGv, we show a set of nodes called a *visible object*.

4.3.1 Types of nodes

A sort of a node in HTGv is represented in color.

A root node represents the root of a hierarchical tree of HTGv and is displayed in red. There is only one root node in a visible object.

A basic node corresponds to a statement in the source code that is executed sequentially in the same iteration because of a data dependence. It is displayed in light blue.

A parallel node represents a loop body that can be executed in parallel with other parallel nodes in the same iteration because there are no data dependences. It is displayed in green.

A loop node represents a loop and is also displayed in red.

A call node represents a function call and is displayed in dark blue.

An if node represents an *if* statement and is displayed in yellow.

Although both root nodes and loop nodes are shown in red, there is only one root node in an HTGv. Therefore we are not confused at a distinction between a root node and loop nodes. The root node and loop nodes have the same properties that both have children in the HTGv.

4.3.2 Program flows

The x-axis represents a program flow, which indicates the order of execution of each node with a number. A node with a big number is executed after the termination of a node with a smaller number.

We use the sequence number of the execution of a node in HTGv as the x coordinates of the node.

4.3.3 Hierarchical levels of loop structure

The z-axis represents the level in the hierarchical structure based on loops. Since we are interested in parallelizing methods related to loops, we focus on the hierarchical structure obtained from HTGv. In this hierarchical structure, each level represents a loop and we can easily discern whether the loop has been parallelized.

When loop node α is included in the body of loop node β, we say α is a deeper node or in a deeper hierarchy than β. Similarly, we say β is a shallower node or in a shallower hierarchy than α.

Values of the z-axis for shallower hierarchies are smaller than those for deeper ones. The top level of the hierarchy, where the value of the z-axis is 0, consists of just a root node. Any nodes in the second level, where the value of the z-axis is 1, do not belong to any loop. We use the hierarchical level of a node in HTGv as the z coordinates of the node.

4.3.4 Measures of parallelism

On the y-axis, nodes are placed with respect to the number that corresponds to a measure of parallelism so that users can intuitively find which parts of the program have been parallelized.

Usually, such a measure of parallelism is given as the number of instructions that can be executed in parallel. Since our target is source-level information, we define our measure of parallelism there.

Most current parallelizing compilers try to parallelize a loop at the iteration level. They try to divide each iteration between different processors and execute them in parallel. A loop whose iterations can be distributed between other processors is usually expressed as a DOALL loop. Therefore, we assume the loops that can be executed in parallel are expressed by DOALLs in the source code.

We define our measure of parallelism as follows:

1. The measure of parallelism at the root node is 1.

2. The measure of parallelism outside of any loops is 1.

3. When the measure of parallelism in a loop that belongs to the hierar-

chical level $n(1 \leq n)$ is $w(1 \leq w)$, the measure of parallelism of the nodes that belong to the body of the loop is as follows:

- $w \times p$, when the loop is DOALL and the total number of the loop iterations is p.
- Otherwise, w.

Program Structure View displays nodes in the y-direction according to the number corresponding to the measure of parallelism. According to this measure, a sequence of nodes that spreads along the y-axis has high parallelism while another sequence of nodes that does not spread along the y-axis is not executed in parallel. Thus, users may pay attention to the part that has no spread along the y-axis. In practice, it is hard to compare the measure of parallelism when loop bounds are given as variables. In this case, the Program Structure View asks users for concrete values for the variables.

To put it concretely, the process of calculating the measure of parallelization is done at the same time as plural nodes of HTGv are made from a node of HTG.

Figure 4.4 shows a simple example of PSV. PSV visualizes a node of HTGv as a colored cube the coordinates of which are decided by the rules explained in this chapter. In addition, PSV displays dotted lines to clarify the relationships between nodes. Users can select visible or invisible for the lines. In this figure, the x axis starts from the upper left and ends at the lower right, the y axis starts at the right and ends at the left, and the z axis goes from up to down. Examples of practical programs are shown in chapter 6.

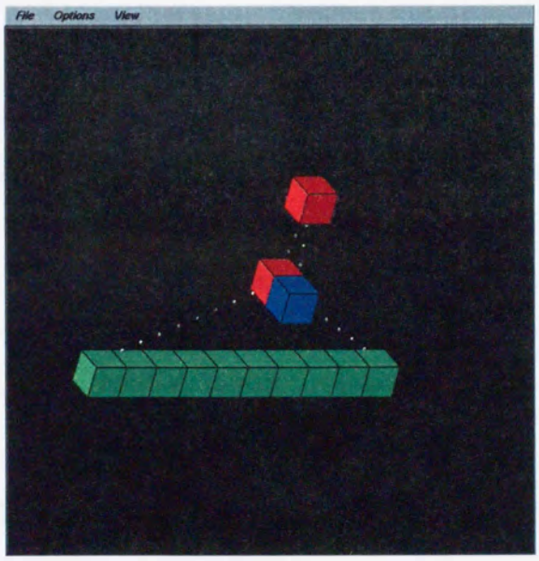
```
      DO 20 I = 1, 10
        A(I) = B(I)
20    CONTINUE
      END
```

(a)

```
Node ID: 14
VISIBLE ID: 1
Type: Parallel
x: 0
y: -4
z: 2
Children: -1
VCHILDREN: -1

Node ID: 14
VISIBLE ID: 2
Type: Parallel
x: 0
y: -3
z: 2
Children: -1
VCHILDREN: -1
```

(b)



(c)

Figure 4.4: A simple example of PSV: (a) is the source program, (b) is a part of HTGv, and (c) is PSV

Chapter 5

Visualization of Data Dependence

In this chapter, we describe the Data Dependence View of NaraView. First, we introduce a data dependence model for the Data Dependence View. It defines data dependence from the point of view of variables. Then, we show how we visualize data dependence based on the model. In the last section, we give an example for clarifying the difference between Banerjee's data dependence model and our data dependence model.

5.1 Variable-Oriented Data Dependence Model

Since data dependence is one of the most important features in a program, there are several models of data dependence for compilation. This section introduces a typical data dependence model defined by Banarjee [3] for par-

allelizing compilers. Then, we define a new data dependence model for visualization and discuss the correspondence between the models.

5.1.1 Banerjee's data dependence model

There are several models of data dependence for compilation [1], [4], [32], and all of them are defined for optimization in compilers. Therefore, they are based on a *sentence* to express data dependence. A sentence is a basic unit for compilers.

The definition of a sentence may differ for each compiler. In some cases, a sentence is a sentence of a programming language, such as Fortran, C. In other cases a sentence is a sentence of an intermediate language of a compiler.

In this section, we introduce Banerjee's model [3] as a typical model of data dependence. Banerjee's model is generally used in parallelizing compilers. The model aims to express data dependence in loops for parallelization, which is very close to our aim.

In Banerjee's model, a sentence is an assignment statement in Fortran, and an *instance* is the execution of a statement. In a loop, one sentence usually has several instances. The definition of Banerjee's data dependence is below:

Definition 5.1 (Banerjee's data dependence) *When there exists an instance S' of statement S and an instance T' of statement T, and S' and T' satisfy the following conditions, we say "statement T depends on statement S".*

1. *S' and T' access the same data D, and at least one of those accesses is a write.*

2. *S' is executed before T'.*

3. *In the same execution, D is not accessed as a write between S' and T'.*

Banerjee also defined three kinds of data dependence:

- If S' accesses D in a write and T' accesses D in a read, the data dependence from S' to T' is a flow dependence.
- If S' accesses D in a read and T' accesses D in a write, the data dependence from S' to T' is an anti dependence.
- If S' accesses D in a write and T' accesses D in a write, the data dependence from S' to T' is an output dependence.

5.1.2 Variable-oriented data dependence model

This section proposes a new data dependence model based on variables. First, we define an *access* to a variable, then we define data dependence as a relationship between accesses. The aim of our data dependence is to express a period in which we must keep the value of a variable (called "lifetime of an attribute" by Aho at el. [1]), that is, we express a period in which we may not keep the value of a variable clearly. Unlike Banerjee's model, we don't restrict our model in a loop. Therefore, our meaning of sentence is different from that of Banerjee. Banerjee defines an assignment statement as a sentence. We use the word "sentence" in our model as a sentence of

Fortran source code. However, we use the word "instance" in the same way as Banerjee.

Below, we define a *reference* to a variable and a *conceptual time* for preparing of the definition of an access. Here, a variable means a variable or an element of an array in a source program.

Definition 5.2 (reference) *A reference to a variable v is expressed as $R(v, t, k)$, in which v is a variable, t is a time, and k is a type of a reference, either read (R) or write (W). We suppose there is at most one reference to v at t, since t is small enough.*

Definition 5.3 (conceptual time) *A conceptual time provides an order of execution of instances in a program. If $t_i < t_j$, instances in t_i are executed before instances in t_j.*

Conceptual time and the time used in the definition of reference are different. A conceptual time may not be as small as the time in the definition of a reference.

We define an access as a set of references by a conceptual time.

Definition 5.4 (access) *An access $A(v, t, k)$ is a set of all references to variable v in conceptual time t, in which k is a type of access: read (R), write (W), or read and write (R&W).*

Read (R) *All references to variable v in conceptual time t are read references.*

Write (W) *The first reference to variable v in conceptual time t is a write reference.*

Read and write (R&W) *The first reference to variable v in conceptual time t is a read reference and there is at least one write reference.*

Various conceptual times exist in various granularity:

- At conceptual time t_i, there is at most one reference. That is the time used in the definition in the reference. We call it *reference time.* This has the smallest granularity of conceptual times.

- At conceptual time t_i, there is at most one write reference, and if there is a write reference at t_i, the write reference is the latest reference in time t_i. For example, references produced by an assignment statement in Fortran are expressed in the same time t_i of this granularity of conceptual time. Therefore, we call this *statement access time.*

- At conceptual time t_i, there are several references, all of which occur in the same iteration in a loop. We call this *iteration access time.*

We can suppose that there is a perfect double loop in which the number of times of the outer loop is n, the number of times of the inner loop is m, and there are k statements in the loop. If we express accesses in this loop by statement access time, we need $n \times m \times k$ number of times. If we use an iteration access time, we need $n \times m$ number of times. If we use an iteration access time for the outer loop, that is, we use iteration as iteration of the outer loop, we need n number of times.

If we use the reference time as conceptual time, each access includes only one reference. Therefore, the type of access is read or write. A type of an access can never be read and write.

If we use the statement access time as conceptual time, the type of an access $A(v, t, k)$ is one of the three types, but it has special meaning. In the case of write access, there is no read reference to v in t. In the case of read and write access, there is more than one read reference to v in t, and the last reference to v in t is a write reference.

If we use the iteration access time as conceptual time, k of an access $A(v, t, k)$ may be one of the three types. In the case of read, there are only read references to v in the iteration at t. In the case of write, the first reference to v in the iteration at t is a write reference: after that, we do not know whether or not there is a reference. In the case of read and write, the first reference to v in the iteration at t is a read reference; after that, there is at least one write reference to v in the iteration at t. This is the most general interpretation of accesses.

We can define variable-oriented data dependence based on accesses.

Definition 5.5 (variable-oriented data dependence) *There are two accesses to variable d:* $A_i(d, t_i, k_i)$ *and* $A_j(d, t_j, k_j)$ $(t_i < t_j)$.

1. *There is W-R dependence from* t_i *to* t_j *on d (in symbols,* $WR(d, t_i, t_j)$*), if the following conditions apply:*

   (a) $(k_i = (W \vee R\&W))$

   (b) $(k_j = (R \vee R\&W))$

(c) There is no $A_k(d, t_k, k_k)$ *such that* $t_i < t_k < t_j, k_k = (W \vee R\&W)$.

2. *There is R-W dependence from* t_i *to* t_j *on* d *(in symbols,* $RW(d, t_i, t_j)$*), if the following conditions apply:*

   (a) $k_j = W$

   (b) There is no $A_k(d, t_k, k_k)$ *such that* $t_i < t_k < t_j$.

Variable-oriented data dependence is defined as a relationship between accesses. Therefore, our variable-oriented data dependence can be used not only in a loop but in all of a program. Since we use conceptual time to define the variable-oriented data dependence, we can use this data dependence in various granularity. We can also define data dependence between accesses that are expressed by different conceptual times, if the conceptual times satisfy Definition 5.3.

If accesses are expressed by a conceptual time in larger granularity than the statement access time, for example iteration access time, a read and write access means that data dependence exists in the access. To investigate the data dependence, we may re-express the access in conceptual time with smaller granularity, such as statement access time.

The most important property of variable-oriented data dependence is the correspondence of the lifetime of an attribute [1] and W-R dependence.

Definition 5.6 (The longest W-R dependence) *The longest W-R dependence is defined to a write access* $A(d, t, k)$ $(k = W)$ *as the following:*

1. *If there is no W-R dependence caused by* $A(d, t, k)$*, there is no longest W-R dependence.*

2. *If there are W-R dependences* $WR(d, t, t_i)(1 \leq i \leq n, n \geq 1)$ *caused by* $A(d, t, k)$, *the longest W-R dependence is* $WR(d, t, t_n)$, *in which if* $j \leq k, t_j \leq t_k$.

The longest W-R dependence is identical to the lifetime of an attribute.

If we use the statement access time as the conceptual time, we can define other kinds of dependence as the following:

Definition 5.7 (W-W dependence) *When there are two accesses to variable d,* $A_i(d, t_i, k_i)$ *and* $A_j(d, t_j, k_j)$, *on statement access time, there is W-W dependence from* t_i *to* t_j *on d (in symbols,* $WW(d, t_i, t_j)$*) if* $k_i = (W \vee R\&W)$ *and* $k_j = W$ *and there is no* $A_k(d, t_k, k_k)$ *in* $t_i < t_k < t_j$.

A W-W dependence $WW(d, t_i, t_j)$ indicates the existence of a wasted write access, since the value written by the write reference in the access $A_i(d, t_i, k_i)$ is never read by any references before the write reference in the access $A_j(d, t_j, k_j)$ rewrites it.

5.1.3 A comparison between Banerjee's data dependence model and the variable-oriented data dependence model

The goal of the variable-oriented data dependence model is to show data dependence to a user for parallelization. Thus, we must show that the variable-oriented data dependence model is powerful enough to be used in parallelization. As we mentioned in section 5.1.1, Banerjee's data dependence model (flow dependence, anti dependence and output dependence) is used in many

parallelizing compilers to check whether a loop can be executed in parallel. Therefore, we show the correspondence of Banerjee's data dependence to our data dependence (W-R dependence, R-W dependence and W-W dependence).

Banerjee's data dependence is defined on Fortran assignment statements in a loop. Our data dependence can be defined in various granularity, but we can easily convert data dependence expressed in one granularity into another, since both are based on the same references. Data dependence on the statement access time has the same granularity as Banerjee's data dependence. Therefore, to show a correspondence of Banerjee's data dependence to our data dependence, we can show the correspondence on statement access time. In this section, all of our data dependences are on statement access time.

Banerjee's data dependence is defined as a relationship between statements. Data dependence between statements is represented by our data dependence model as follows:

$$\forall d$$

$$\{WR(d, S', T') | WR(d, S', T') \in DEP\} \cup$$

$$\{RW(d, S', T') | RW(d, S', T') \in DEP\}$$

in which, S' and T' are instances of S and T, respectively, and DEP is the set of data dependence.

Below we show the correspondence of Banerjee's flow dependence, anti dependence and output dependence with our W-R dependence and R-W dependence.

Theorem 5.1 *A set of flow dependence on sentence S and T is identical to a set of W-R dependence on sentence S and T.*

Proof 5.1 *The proof is found in definition 5.5.*

Theorem 5.2 *An R-W dependence is an anti dependence, an output dependence, or both.*

Proof 5.2 *According to the definition, there is R-W dependence iff there are consecutive accesses A_i, A_j, and (k_i, k_j) of A_i, A_j are (R, W), (W, W), or $(R\&W, W)$, respectively. If (k_i, k_j) is (R, W), the dependence is an anti dependence. If (k_i, k_j) is (W, W), the dependence is an output dependence. If (k_i, k_j) is $(R\&W, W)$, the dependence is both an anti dependence and an output dependence. But even if a dependence is an output dependence, in which k_i, k_j is $W, R\&W$, the dependence is not an R-W dependence. If a dependence is an anti dependence, in which k_i, k_j is $R, R\&W$, the dependence is not an R-W dependence.*

Theorem 5.3 *A set of W-W dependence is a subset of output dependence.*

Proof 5.3 *According to the definition, if a dependence is a W-W dependence, it is an output dependence. But the reverse is not true. For example, there are three consecutive accesses: write-read-write. There is an output dependence from the first access to the third access, but there is no W-W dependence.*

Figure 5.1 shows the correspondence and difference between Banerjee's data dependence and our variable-oriented data dependence. The advantage

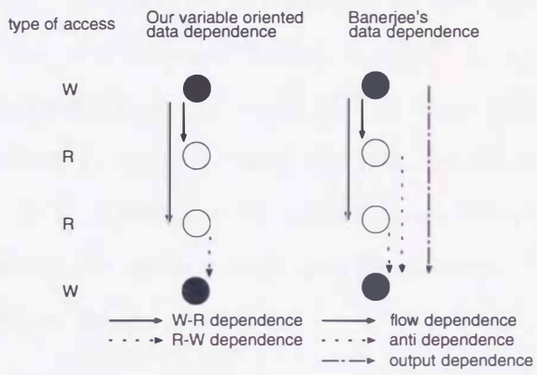


Figure 5.1: Our variable-oriented data dependence model and Banerjee's data dependence model

of Banerjee's data dependence is that we can judge whether we change an execution order of two statements just by checking data dependence between the two statements. For optimization and parallelization, it is common to change an execution order of statements unless their data dependence is unchanged. As for our variable-oriented data dependence, R-W dependence and W-W dependence represent only a dependence between consecutive accesses. Therefore, we cannot decide to change the execution order of two statements based only on the data dependence of the statements. If we visualize all data dependence, however, we can judge it at a glance.

The advantage of our data dependence model is that we can know the period in which we must keep the value of a variable as a W-R dependence. R-W dependences and W-W dependences let us know the period in which we cannot keep the value of a variable. In Banerjee's data dependence model, how-

ever, anti dependence and output dependence do not show the period. For example, if there are three accesses $A_i(d, t_i, k_i)$, $A_k(d, t_k, k_k)$, and $A_j(d, t_j, k_j)$ ($t_i < t_k < t_j$), in which $(k_i, k_k, k_j) = (R, R, W)$, there is an anti dependence from the statement A_i to the statement A_j. But the value of d must remain the same from t_i to t_k. Therefore, we cannot know the period in which we cannot keep the value of a variable from anti dependence. The same problem exists in output dependence.

5.2 Data Dependence View

In this section, we explain how we visualize data dependence expressed by our variable-oriented data dependence model in the Data Dependence View (DDV). The DDV visualizes accesses and dependences in a loop in three-dimensional graphics.

5.2.1 Accesses

An access is displayed as a colored cube. The coordinates of each cube are expressed by a triplet (x,y,z). Each axis and color has the following meaning:

x,y	location of the data
z	time
color	the type of the access

The layout of data on the x-y plane is defined by the user. The *AVD map*, which is described in section 5.2.3, shows the current data layout.

Table 5.1: Colors of poles

type	color
W-R pole	green
R-W pole	yellow
W-W pole	pink

Time is a value of the conceptual time. In the current version of NaraView, the user can select the statement access time or iteration access time on each loop nest.

The type of access is represented in a color:

A blue cube represents a read access.

A red cube represents a write access.

A purple cube represents a read and write access.

5.2.2 Dependence

Data dependence is represented as a colored pole that connects two cubes and indicates accesses to the same data. There are three kinds of poles that correspond to W-R dependence, R-W dependence, and W-W dependence. We call these the W-R pole, R-W pole, and W-W pole, respectively. They are distinguished from each other by the colors listed in Table 5.1.

According to the definition of data dependence, poles of different types do not share the same coordinates. In three-dimensional graphics, however, ac-

cording to the viewpoint, some poles overlap other poles in other coordinates, so the user can select visible or invisible for the following five items:

- W-R dependence
- The shortest W-R dependence
- R-W dependence
- The shortest R-W dependence
- W-W dependence

Looking for the shortest dependence is useful for the user to investigate a way to parallelize a given program (see chapter 6). Therefore, we provide the items needed to show only the shortest W-R dependence or R-W dependence.

5.2.3 Other indicated objects

AVD map

A plane, the array-variable disposition (AVD) map, is placed perpendicular to the z-axis. It shows the layout of arrays and variables and allows the user to easily comprehend the view of data dependence. Characters on the AVD map stand for the names of variables or arrays that are mapped there.

A user can define the layout of variables on the AVD map. Usually, we assign one variable or one element of an array to one place, but we can also assign some variables or some elements of arrays (for example, all elements of one column of an array) to one place, and the user can select visible or invisible for the AVD map.

Loop grids

Loop grids are semi-transparent planes that are placed perpendicular to the z-axis. Users can display loop grids to indicate the beginning of each iteration of a loop or specified iterations.

Loop grids are important for showing the relationships between data dependence and the given source code. For example, suppose there is a double nested loop. If users specify to show loop grids at each beginning of the inner loop, they can find the pattern caused by the outer loop at a glance.

As we will mention the next section, loop grids play an impotrant role in investigating that a loop has more parallelism or not. Concreate examples are shown in chapter 6.

5.2.4 An interpretation

From a figure generated by DDV, we can obtain useful information for optimization and parallelization:

- We can find unexpected accesses by poles and cubes.
- If there is a W-W pole on the statement access time, we know the access of the start of the W-W pole is useless as a write access.
- If there is no pole across the loop grid of $z = i$, the before part of the loop grid (the part of $z < i$) and the after part of the loop grid (the part of $z \geq i$) have no data dependence on each other, so the order of execution does not matter.

```
      do 20 i = 3, 10
        do 18 j = 5, 10
          m = a(i, j) - b(i, j)  : S1
          a(i, j) = b(i-3, j-5)  : S2
          b(i, j) = a(i-2, j-4)  : S3
          c(i, j) = m - b(i, j)  : S4
18      continue
20    continue
```

Figure 5.2: Sample program for a comparison

- If no pole starts and ends during $i < z < j$, the accesses in the period can be executed in parallel.
- If we regard an AVD map as a mapping of variables in the memory in an execution, we can examine ways of distributing data using DDV.

5.3 An example for a comparison

In this section, we show an example to compare Banarjee's model and our model. Examples for practical use are explained in detail in chapter 6.

Figure 5.2 is a sample program shown in [22], but we added the variable m. The data dependences of the program in Banerjee's model are listed in Table 5.2. The dependences of 1, 4 and 8 are on variable m, 2 and 5 are on array a, and 3, 6 and 7 are on array b.

Table 5.2: The data dependences of the sample program (Banerjee)

1. An output dependence from S_1 to S_1
2. An anti dependence from S_1 to S_2
3. An anti dependence from S_1 to S_3
4. A flow dependence from S_1 to S_4
5. A flow dependence from S_2 to S_3
6. A flow dependence from S_3 to S_2
7. A flow dependence from S_3 to S_4
8. An anti dependence from S_4 to S_1

Table 5.3: The variable-oriented data dependences of our sample program

$$
\begin{aligned}
&WR(m, S_1(3,5), S_4(3,5)), \ldots, WR(m, S_1(10,10), S_4(10,10)) \\
&RW(m, S_4(3,5), S_1(3,6)), \ldots, RW(m, S_4(10,9), S_1(10,10)) \\
&RW(a(3,5), S_1(3,5), S_2(3,5)), WR(a(3,5), S_2(3,5), S_3(5,9)) \\
&RW(a(3,6), S_1(3,6), S_2(3,6)), WR(a(3,6), S_2(3,6), S_3(5,10)) \\
&\ldots \\
&RW(b(3,5), S_1(3,5), S_3(3,5)), WR(b(3,5), S_3(3,5), S_4(3,5)), \\
&WR(b(3,5), S_3(3,5), S_2(6,10)) \\
&RW(b(3,6), S_1(3,6), S_3(3,6)), WR(b(3,6), S_3(3,6), S_4(3,6)) \\
&\ldots \\
&RW(b(10,10), S_1(10,10), S_3(10,10)), WR(b(10,10), S_3(10,10), S_4(10,10))
\end{aligned}
$$

Data dependences in our model are shown in Table 5.3. Figure 5.3 shows its DDV. The x axis starts from the upper left and ends at the lower right, the y axis starts at the lower left and ends at the upper right, and the z axis goes from down to up. We show loop grids that correspond to the outer loop. W-R poles are displayed. Variables and arrays are mapped from the left in the order of array a, b, variable m, and array c.

This example shows Banerjee's model is not intuitive for users. Our model is also not intuitive in text form, because there are too much information. But when we visualize our model, the information is arranged clear and users can easily understand the pattern of data dependence.

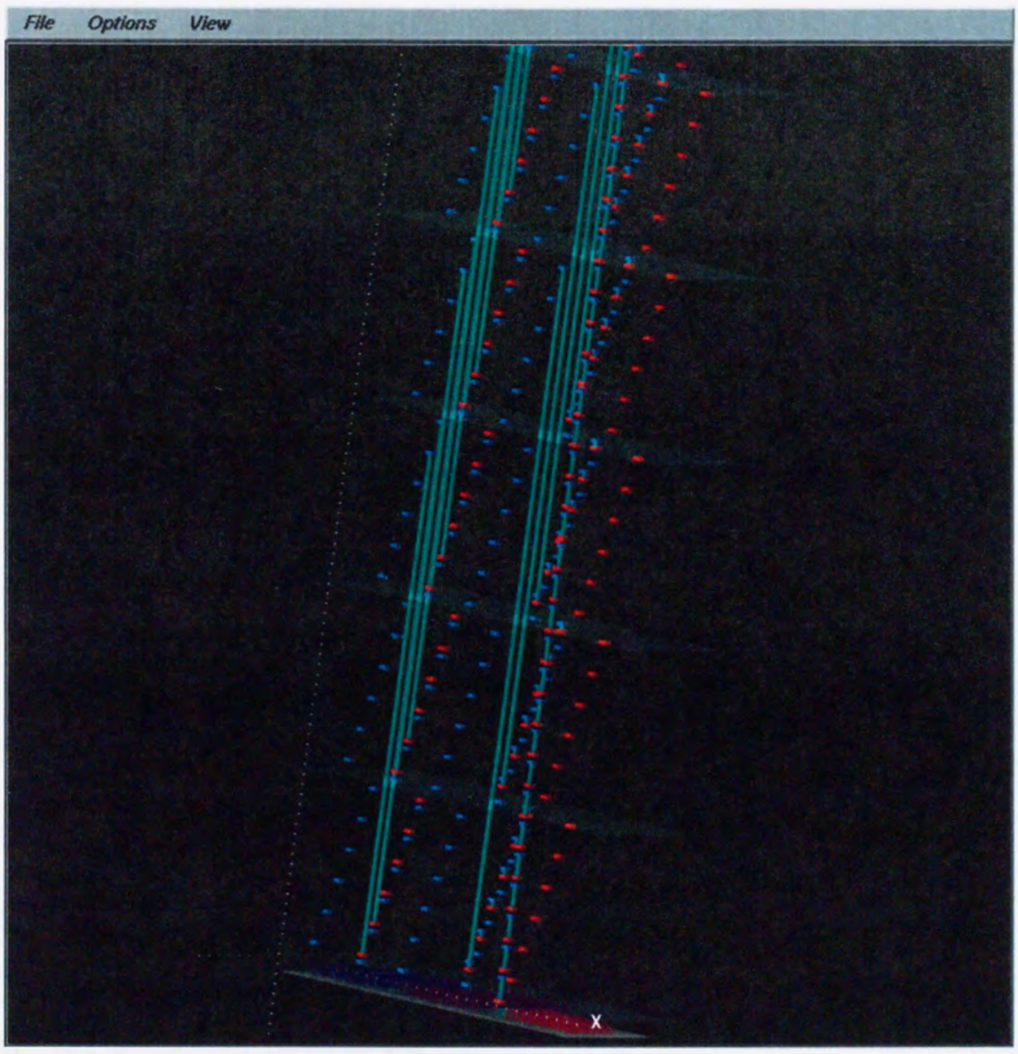


Figure 5.3: A Data Dependence View of the sample program

Chapter 6

Examples

This chapter contributes various examples of NaraView. In the first section, we explain an implementation of NaraView. In the second section, we show typical way to use NaraView: to compile a program, see PSV, investigate a part of the program with DDV and SCV, recompile the program, and see PSV again to check the effect of reconstructing the program. In the third and forth sections, we show characteristic figures generated by PSV and DDV, and we explain how we can get information about reconstructing a program from these figures.

6.1 Implementation

NaraView is implemented in C language on X Windows using OpenGL and Motif. It is divided into modules. Figure 6.1 shows the relationship between these modules.

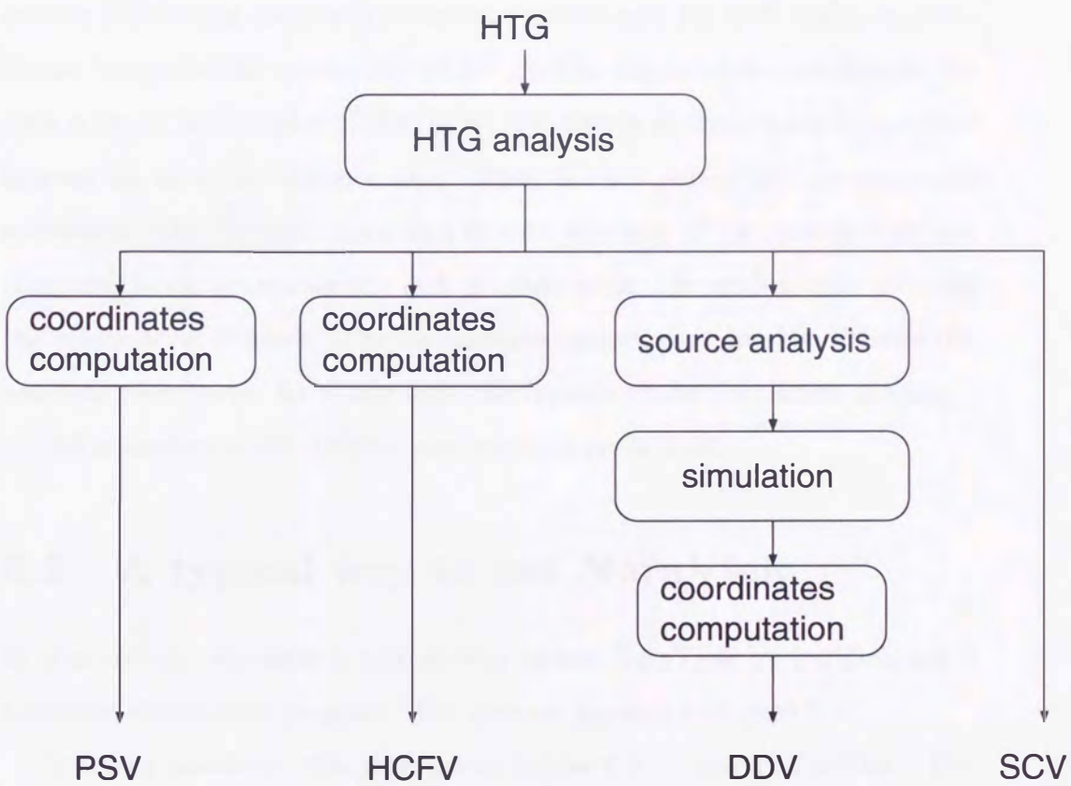


Figure 6.1: Modules of NaraView

There are six modules for analysis and computing coordination and four modules for graphics, which are represented as PSV, HCFV, DDV, and SCV in the figure. First, NaraView analyzes HTG. For SCV, it needs no further process. For PSV, we need the coordinates of the computation module that creates HTGv and decides the concrete coordinates for each node. A coordinate computation module for HCFV decides the concrete coordinates for each node of HTG directly. For DDV, the source analysis module analyzes indexes for all references in a loop. Then, the simulation module computes a concrete value for each index and reports accesses. If the module does not compute the value because of a lack of information, the module asks users for the information it needs. The coordinates computation module decides the concrete coordinates for access from the reports of the simulation module.

All examples in this chapter are executed on SGI O2.

6.2 A typical way to use NaraView

In this section, we show a typical way to use NaraView by parallelizing a Gaussian elimination program. The process appears in Figure 6.2.

We then parallelize the program in Figure 6.3 by using NaraView. The number at the end of each line in Figure 6.3 indicates the line number generated by Parafrase-2. Parafrase-2 and NaraView refer to the code by the line number.

First, we compile the program with Parafrase-2 with default passes. By default passes, only the loops that have no data dependence are parallelized. Figure 6.4 represents the PSV of an automatically parallelized program cre-

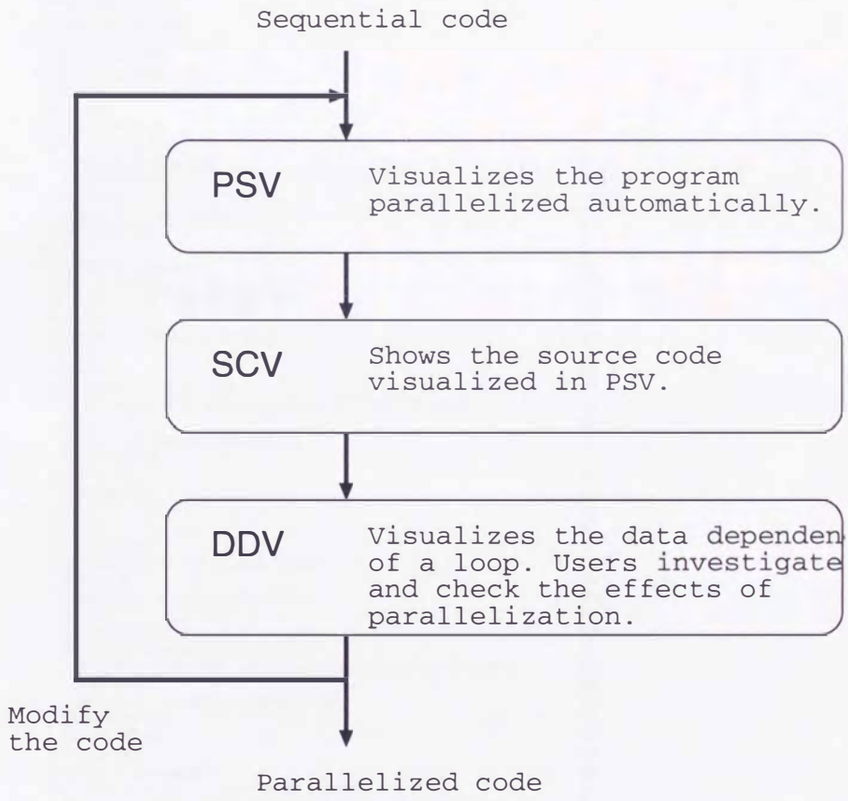


Figure 6.2: A typical use of NaraView

```
      SUBROUTINE gaus(xary,n)
      IMPLICIT NONE
      INTEGER n3, n2, i1, n1, intvl1, k0, intvl0
      INTEGER i0, n0
      INTEGER maxn
      PARAMETER(maxn = 100)
      INTEGER np
      PARAMETER(np = 16)
      INTEGER n
      INTEGER pivot, i, j, k, kk, begtim, endtim
      INTEGER delta, intvl
      INTEGER idx(100)
      INTEGER p(100)
      REAL maxary(100)
      REAL maxelm, sum, m
      REAL xary(100)
      REAL aary(100,100)
      REAL tary(100,100)
      REAL err
      DO 200 i = 1,n                                                    1
        DO 100 j = 1,n                                                  2
          tary(i,j) = aary(i,j)                                         3
100       CONTINUE                                                      4
        idx(i) = i                                                      5
        tary(i,n + 1) = aary(i,n + 1)                                   6
200     CONTINUE                                                        7
      n2 = n                                                            8
      DO 2000 i = 1,n2 - 1                                              9
        intvl = int((n - i) / 16) + 1                                   10
        IF (intvl .GE. 100) GOTO 987                                    11
        pivot = i                                                       13
        maxelm = abs(aary(idx(i),i))                                    14
        i0 = i                                                          15
        n0 = n                                                          16
        DO 1200 k = i0 + 1,n0                                           17
          IF (maxelm .GE. abs(aary(idx(k),i))) GOTO 1200                18
          pivot = k                                                     20
          maxelm = abs(aary(idx(k),i))                                  21
1200      CONTINUE                                                      22
        GOTO 789                                                        23
987     CONTINUE                                                        24
        i1 = i                                                          25
        n1 = n                                                          26
        intvl1 = intvl                                                  27
        DO 1400 k = 0,( -i1 + n1) / intvl1                              28
          p(k) = k                                                      29
          maxary(k) = abs(aary(idx(k),i))                               30
          k0 = k                                                        31
          intvl0 = intvl                                                32
          DO 1300 kk = k0,intvl0 + k0 - 1                               33
            IF (maxary(k) .GE. abs(aary(idx(kk),i))) GOTO 1300          34
            p(k) = kk                                                   36
            maxary(k) = abs(aary(idx(kk),i))                            37
1300        CONTINUE                                                    38
1400      CONTINUE                                                      39
        pivot = p(i)                                                    40
        maxelm = maxary(i)                                              41
        DO 1500 k = 0,( -i - intvl + n) / intvl                         42
          IF (maxary(i + intvl + intvl * k) .LE. maxelm) GOTO 1500      43
          pivot = p(i + intvl + intvl * k)                              45
          maxelm = maxary(i + intvl + intvl * k)                        46
1500      CONTINUE                                                      47
789     CONTINUE                                                        48
        IF (pivot .EQ. i) GOTO 345                                      49
        k = idx(pivot)                                                  51
        idx(pivot) = idx(i)                                             52
        idx(i) = k                                                      53
345     CONTINUE                                                        54
        DO 1600 k = i + 1,n                                             55
          m = aary(idx(k),i) / aary(idx(i),i)                           56
          DO 1550 j = i + 1,n + 1                                       57
            aary(idx(k),j) = aary(idx(k),j) - m * aary(idx(i),j)        58
1550        CONTINUE                                                    59
1600      CONTINUE                                                      60
2000    CONTINUE                                                        61
```

Figure 6.3: A Gaussian elimination program.

```
      DO 4000 i = n,1, -1                                              62
        xary(i) = aary(idx(i),n + 1) / aary(idx(i),i)                  63
        DO 3400 j = i - 1,1, -1                                        64
          aary(idx(j),n + 1) = aary(idx(j),n + 1) - xary(i) * aary(idx(  65
     1      j),i)                                                      65
3400    CONTINUE                                                       66
4000    CONTINUE                                                       67
      delta = endtim - begtim                                          68
      err = 0                                                          69
      n3 = n                                                           70
      DO 6000 i = 1,n3                                                 71
        sum = 0                                                        72
        DO 5400 j = 1,n                                                73
          sum = sum + tary(i,j) * xary(j)                              74
5400    CONTINUE                                                       75
        IF (abs(sum - tary(i,n + 1)) .GT. 0.001) err = 1               76
6000    CONTINUE                                                       78
      RETURN                                                           79
      END                                                              80
```

Figure 6.3: A Gaussian elimination program (cont.).

ated by Parafrase-2. Labels and arrows have been added for explanation.

In the figure, the flow of the program (the x-axis) starts from the upper left and ends at the lower right, and the measure of parallelism (the y-axis) is expressed as the horizontal width, and the level of hierarchical structure (the z-axis) is indicated as approximately up to down. The dotted lines indicate the axes.

In Figure 6.4, we can easily find two loops that can be executed in parallel. Since those parallelized loops have no data dependence, Parafrase-2 could parallelize them without the user's assistance. The code of one of the parallelized loops, which is indicated as "A Parallelized Loop", is taken from lines 1 to 7 in Figure 6.3. The code of another, which is indicated as "Loop 3", comes from lines 55 to 60, but only the inner loop could be parallelized. Other loops could not parallelized because they had data dependence.

By viewing the PSV, we can guess which loops may be difficult to parallelize and which loops may have a possibility of parallelization. For example, an if-statement is one of the obstacles to parallelization. Therefore, we may

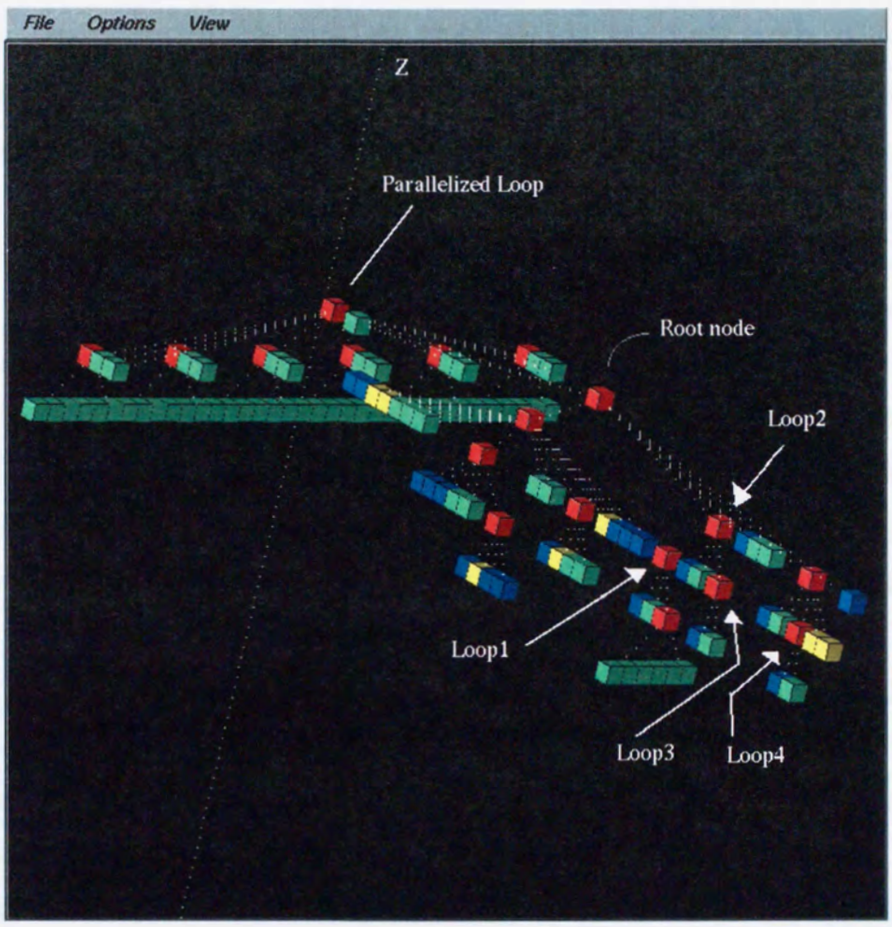


Figure 6.4: A Program Structure View of a program representing Gaussian elimination. The flow of the program (the x-axis) starts from the upper left and ends at the lower right, and the measure of parallelism (the y-axis) is expressed as the horizontal width, and the level of hierarchical structure (the z-axis) is indicated as approximately up to down. The dotted lines indicate the axes.

investigate the possibility of parallelizing the loops that have no if-statements. A function call is also an obstacle. If we want to parallelize a loop that includes a function call, we must have interprocedure analysis of which type of analysis costs more.

The loops that have a possibility of parallelization are indicated in the figure by arrows. Next, we investigate the loops with DDV and SCV.

When we try to see DDV on loop 1 in Figure 6.4, NaraView informs us that there are indirect accesses by array idx in the loop. We know that the indirect accesses in the algorithm of Gaussian elimination do not cause data dependence. Therefore, we instruct Parafrase-2 that there is no data dependence caused by indirect accesses by array idx and compile the program again. Figure 6.5 shows a PSV after second compilation. Loop 3 is parallelized automatically, but loop 1, 2 and 4 are still not parallelized.

Then, we investigate the data dependence in loop 1 by DDV. The source code of this loop is represented in Figure 6.6. The indirect accesses in the loop are ignored as we regard $idx(i)$ as i. This figure displays data accesses when $n = 5$ and $i = 1$. Loop grids that correspond to the outer loop are displayed. The AVD map is placed at the bottom of the figure. The z-axis goes from down to up, which is shown by a dotted line. The inner loop is already parallelized automatically.

Figure 6.7 and 6.8 shows that the data dependence caused by variable m disturbs automatic parallelization of the loop. The W-R poles shown in Figure 6.7 disturb parallelization of the inner loop and the R-W poles shown in Figure 6.8 disturb the parallelization of the outer loop. We can remove the data dependence by scalar expansion on m. The details of this procedure

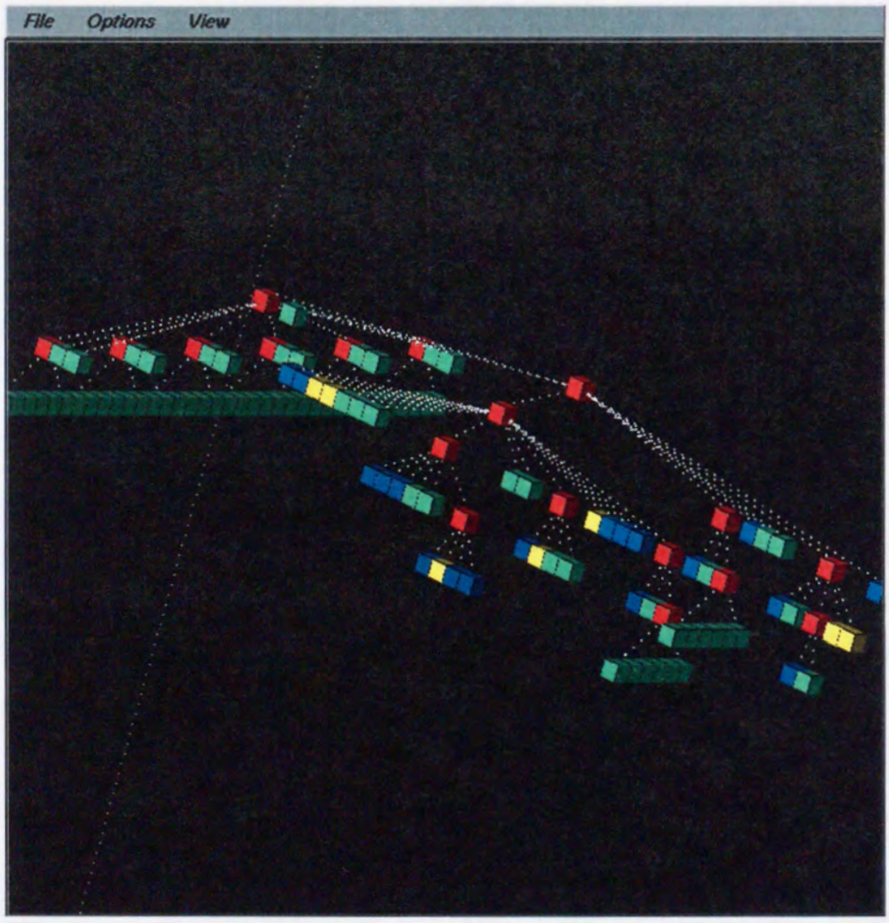


Figure 6.5: A Program Structure View of the Gaussian elimination program after second compilation.

```
      do 1600 k = i + 1, n
       m = aary(idx(k), i) / aary(idx(i), i)
       do 1550 j = i + 1, n + 1
         aary(idx(k), j) = aary(idx(k), j) - m * aary(idx(i), j)
1550   continue
1600  continue
      end
```

Figure 6.6: The source code of loop 1 in the Gaussian elimination program

are explained in section 6.4.3.

Loop 2 has data dependence that we don't know how to remove (Figure 6.9). W-R dependences of array *aary* disturb the parallelization of the outer loop (because the W-R poles of array *aary* go across the loop grids), and W-R dependences of array *xary* disturb the parallelization of the inner loop (because the W-R poles of array *xary* do not go across the loop grids). So, we cannot parallelize the loop. Loop 4 cannot be parallelized because of the data dependence of variable *s* (Figure 6.10). The source code of loop 4 computes the summation of $tAray \times xAray$ (Figure 6.11).

We compile the program again, but loop 1 is still not parallelized. Parafrase-2 reports that there is data dependence on *aary*, but we cannot find the data dependence in DDV (Figure 6.12). So we can confirm removal of the data dependence on m. Each element of the array m (which is produced by scalar expansion) has only one write access, and read accesses to the element occur

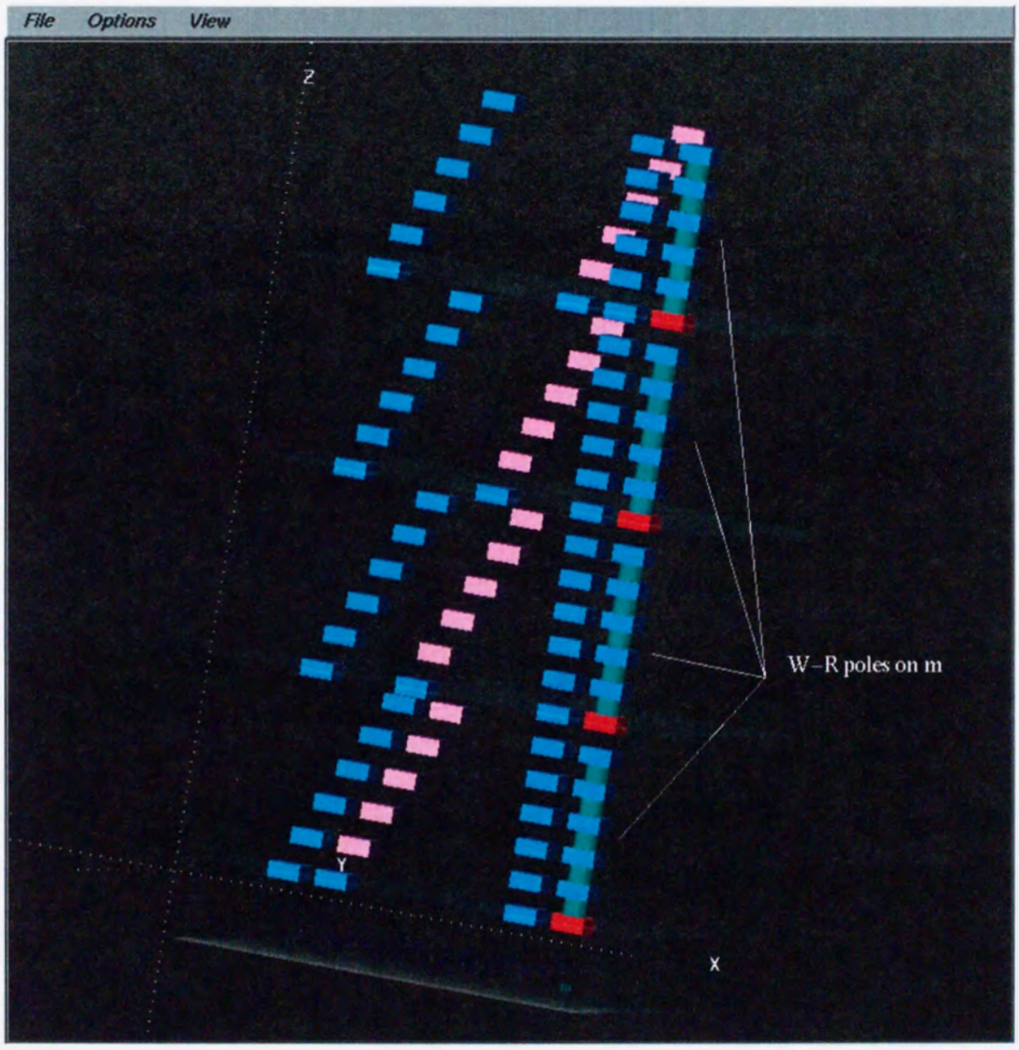


Figure 6.7: A Data Dependence View of loop 1 in the Gaussian elimination program (with W-R poles)

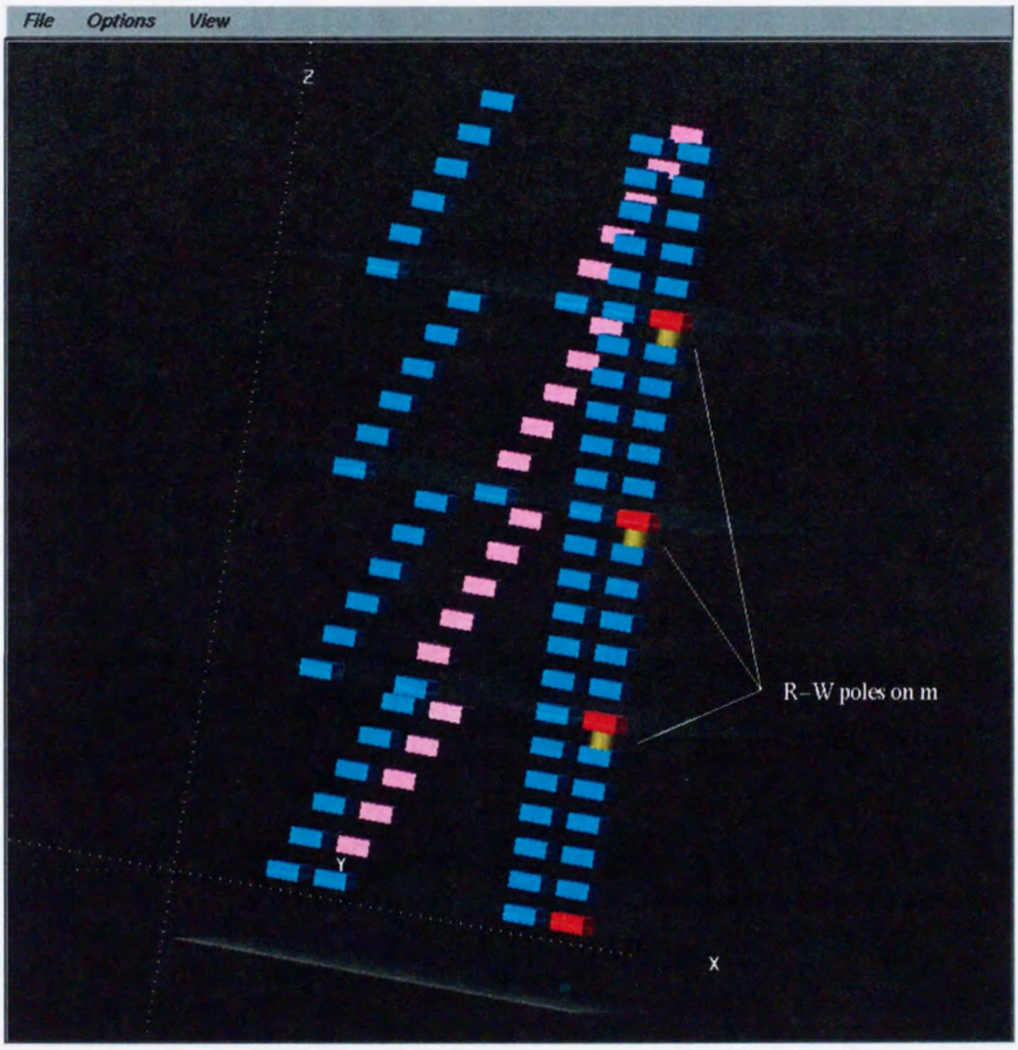


Figure 6.8: A Data Dependence View of loop 1 in the Gaussian elimination program (with R-W poles)

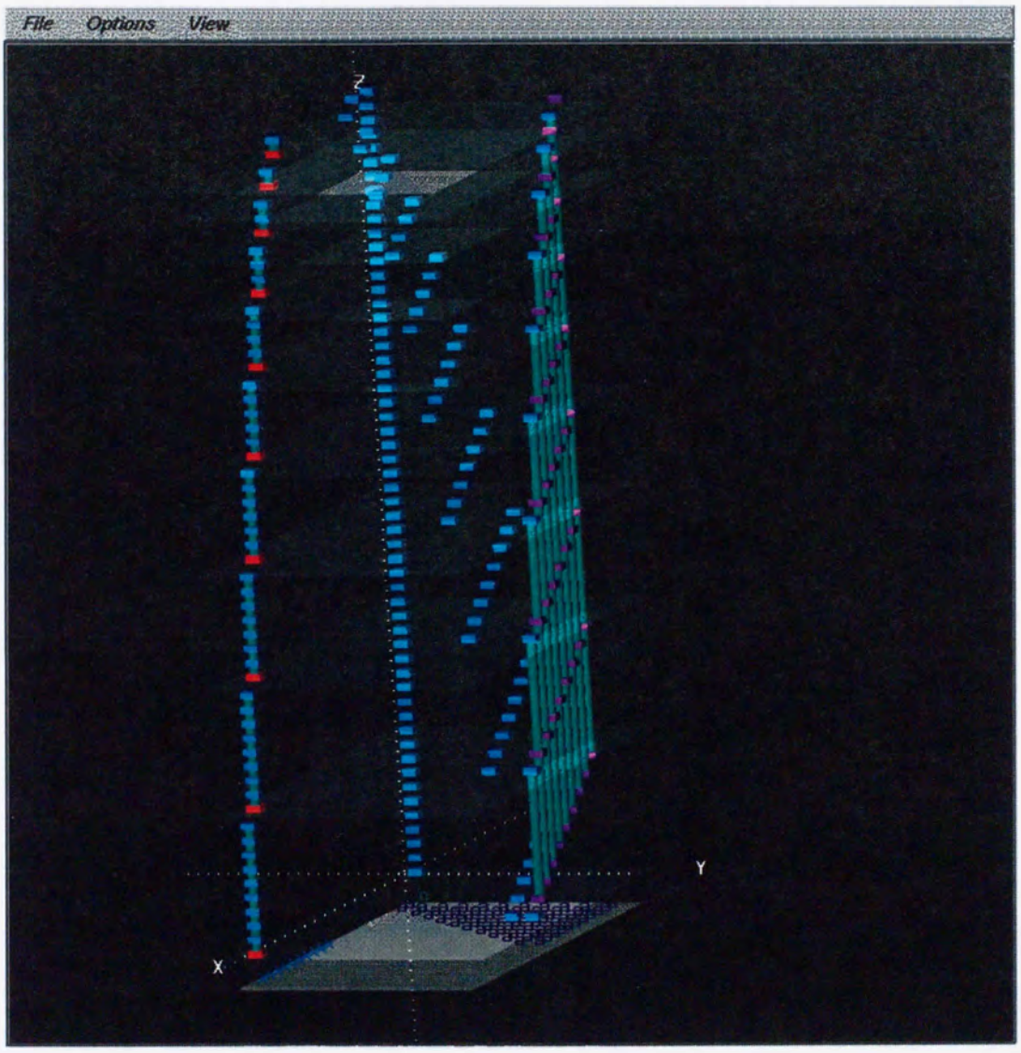


Figure 6.9: A Data Dependence View of loop 2 in the Gaussian elimination program.

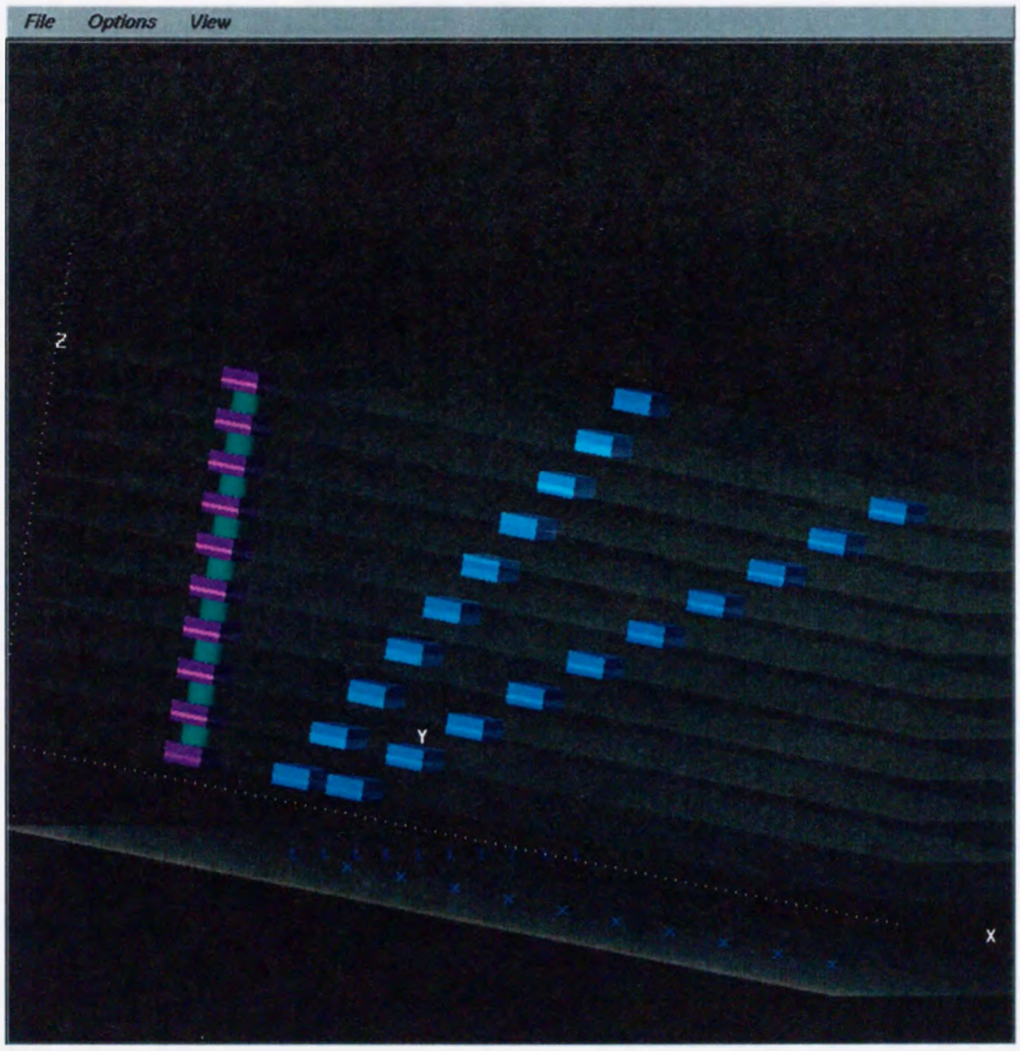


Figure 6.10: A Data Dependence View of loop 4 in the Gaussian elimination program.

```
      do 5400 j = 1, n
        sum = sum + tAry(i, j) * xAry(j)
5400  continue
```

Figure 6.11: The source code of loop 4 in the Gaussian elimination program

in the same region partitioned by the loop grids. And we cannot find data dependence in *aary* in the figure. As we investigate the source code of the loop (Figure 6.6), we find there is no data dependence on *aary* because of the bounds of k and j. Thus, we instruct the compiler that the outer loop can be executed in parallel.

The result of an investigation like this one is a parallelized program. A PSV of the program is shown in Figure 6.13.

6.3 Examples of PSV

This section shows two examples of PSV. The first includes function calls; the second is a "big" program.

If a program includes a function call, the call is represented as a dark blue cube as in Figure 6.14. In this figure, we find 17 function calls. In this version of NaraView, a function call means a CALL statement in Fortran. Statements that invoke functions in other ways, for example, the max function invoked by $m = max(a, b)$, are represented as a basic node. If we find a function call in a program, we may choose giving up parallelization of the loop or performing interprocedure analysis if possible.

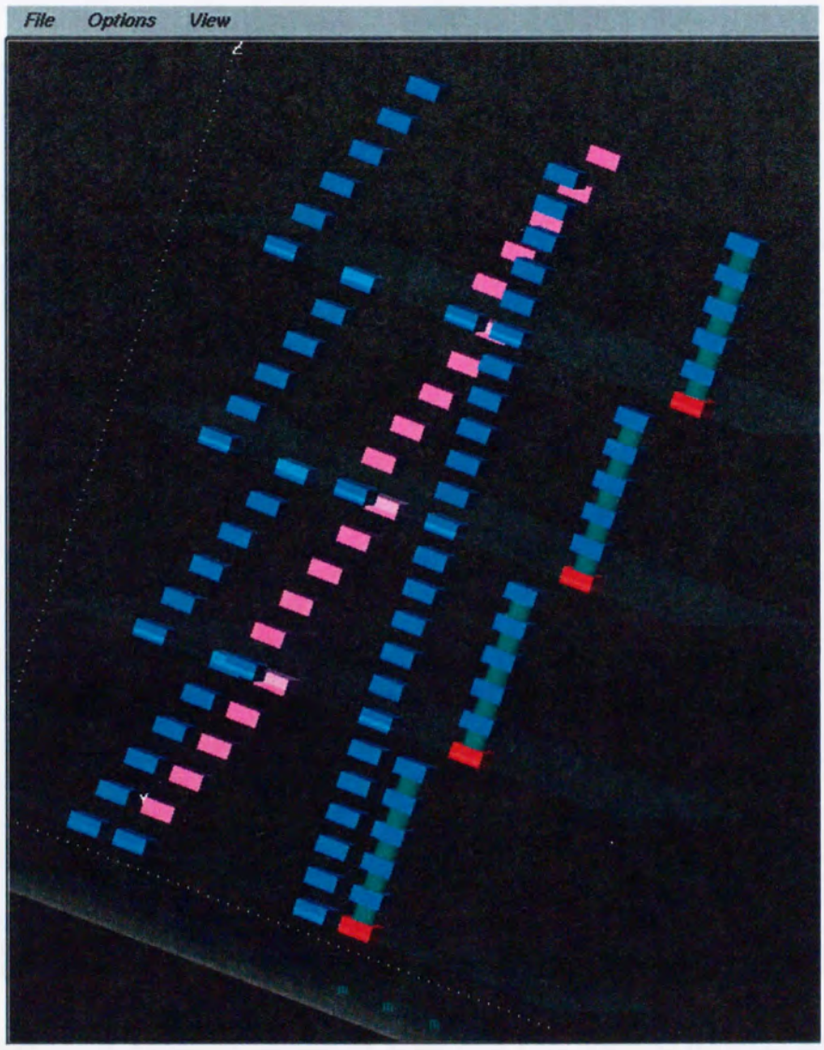


Figure 6.12: A Data Dependence View of loop 1 of the Gaussian elimination program with scalar expansion.

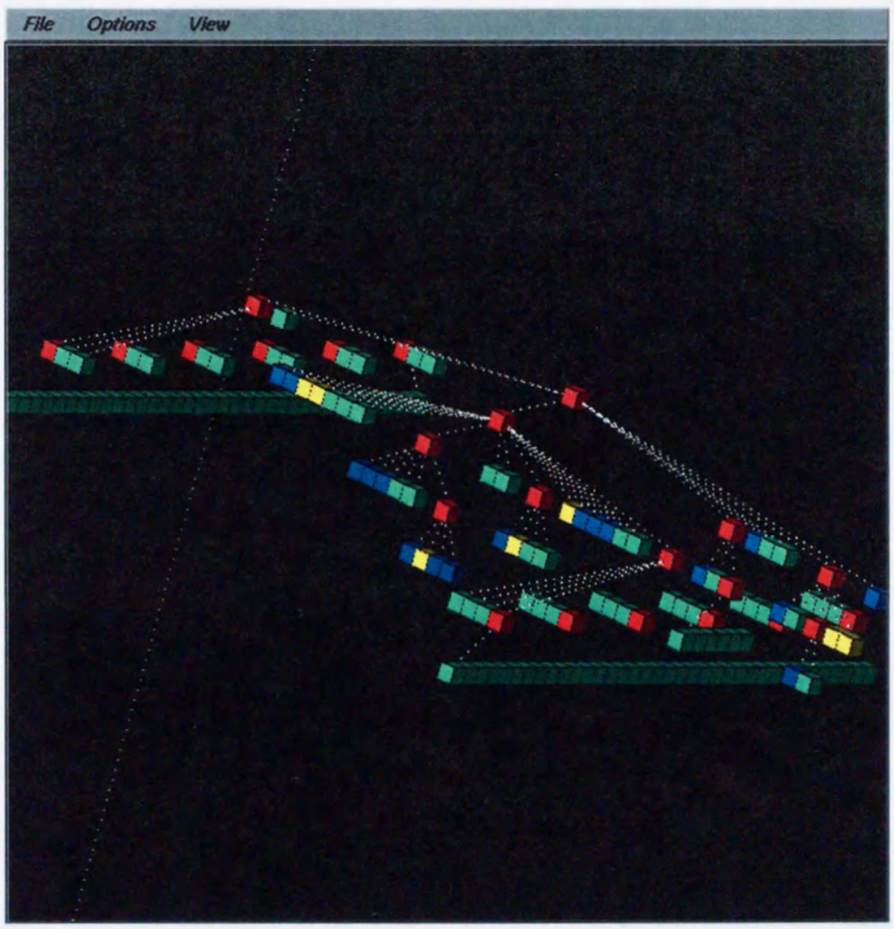


Figure 6.13: A Program Structure View of a parallelized Gaussian elimination program.

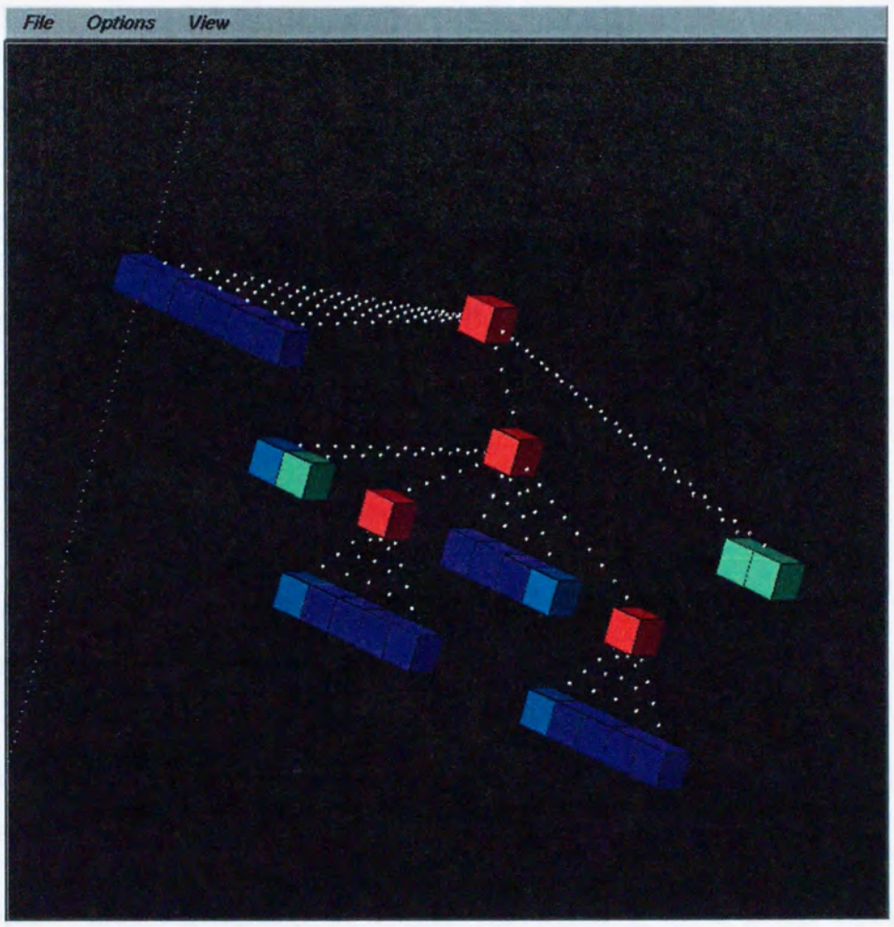


Figure 6.14: A Program Structure View of a program that includes function calls.

The second example is a "big" program. Figure 6.15 is a PSV of the livermore kernel, a well-known benchmark program for automatic parallelization. It consists of 25 independent loops (24 for the benchmark and 1 to report the results) and about 350 lines. We can see the whole visible object in a window and see which loops can be parallelized at a glance. If we want to see each loop, we can scale the object up and rotate it easily. Some researchers comment that program visualization is not suitable for dealing with large programs. But, as we will mention in section 6.5, we think PSV has enough potential to visualize large programs. Because essentially PSV visualizes a function (or subroutine) of a program and the data of PSV have hierarchical structure.

6.4 Examples of DDV

This section contains three examples of the Data Dependence View. The first is an example of cycle shrinking and loop interchange. The second is an example of loop skewing, and the third is a scalar expansion [22]. These examples show how the Data Dependence View helps users select or compare transformation methods.

6.4.1 An example of cycle shrinking and loop interchange

We can show whether or not we decide to apply cycle shrinking by looking at DDV. Cycle shrinking can be applied to any loops (see section 2.1.1). Therefore, we have to judge whether it is useful to apply cycle shrinking.

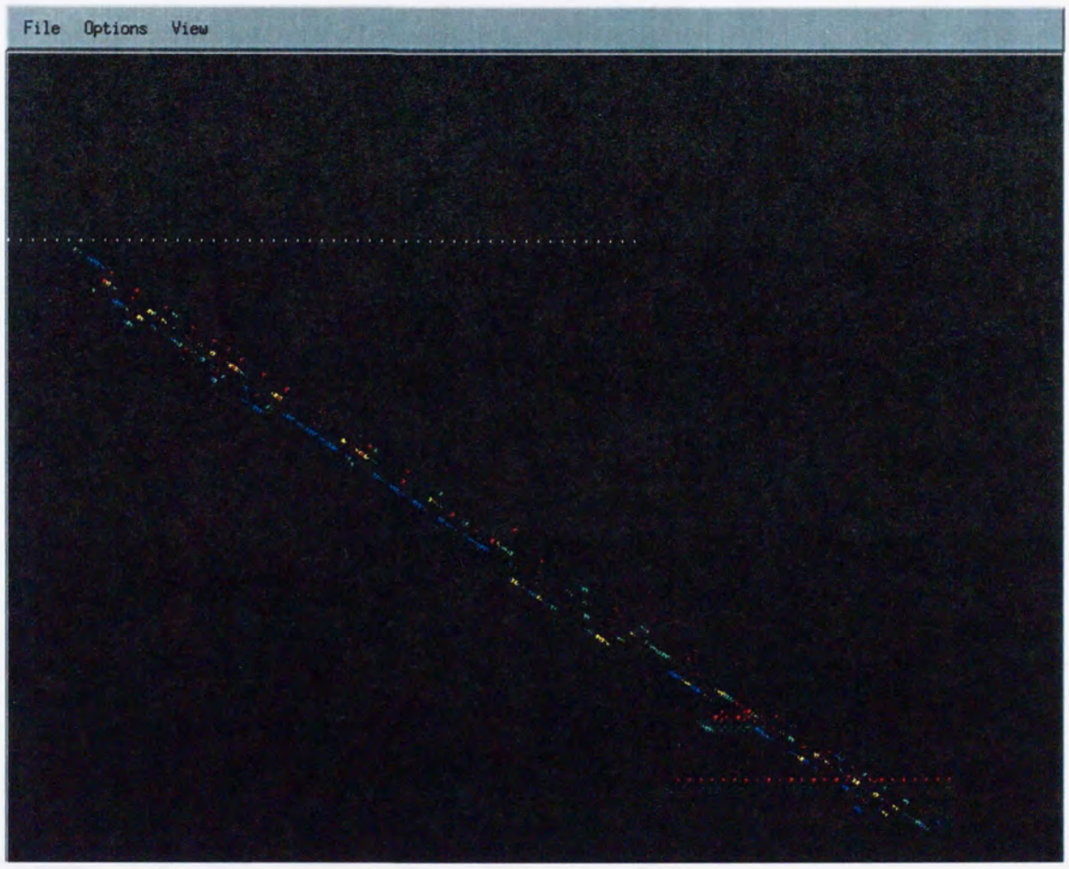


Figure 6.15: A Program Structure View of the livermore kernel.

```
      do 20 i = 3, 10
        do 18 j = 5, 10
          a(i, j) = b(i-3, j-5)
          b(i, j) = a(i-2, j-4)
18      continue
20    continue
```

Figure 6.16: A sample program of cycle shrinking

Figure 6.16 shows a sample program. A DDV of the program is shown in Figure 6.17. The elements of array b are shown in the left in the figure and the elements of array a are in the right.

Figure 6.17 shows there are data dependences in this program. Therefore we cannot parallelize the whole of the program. But, in figure 6.17, the length of W-R poles on array a or b is longer than the intervals of loop grids. This is a sign that we should consider applying cycle shrinking to either of the loops. We are trying to parallelize loops by reconstructing the loops in the source level. Intervals of loop grids represent the number of executions that can be performed in parallel. W-R poles which are longer than the intervals indicate that a loop has the potential for parallelism.

In general, since the overhead of loop dividing in an execution is high, we prefer to reduce the number of dividing. Therefore we would like to parallelize outer loop than inner loop. Then we try to apply cycle shrinking to the outer loop. The result source is shown in Figure 6.18. Figure 6.19 shows a DDV

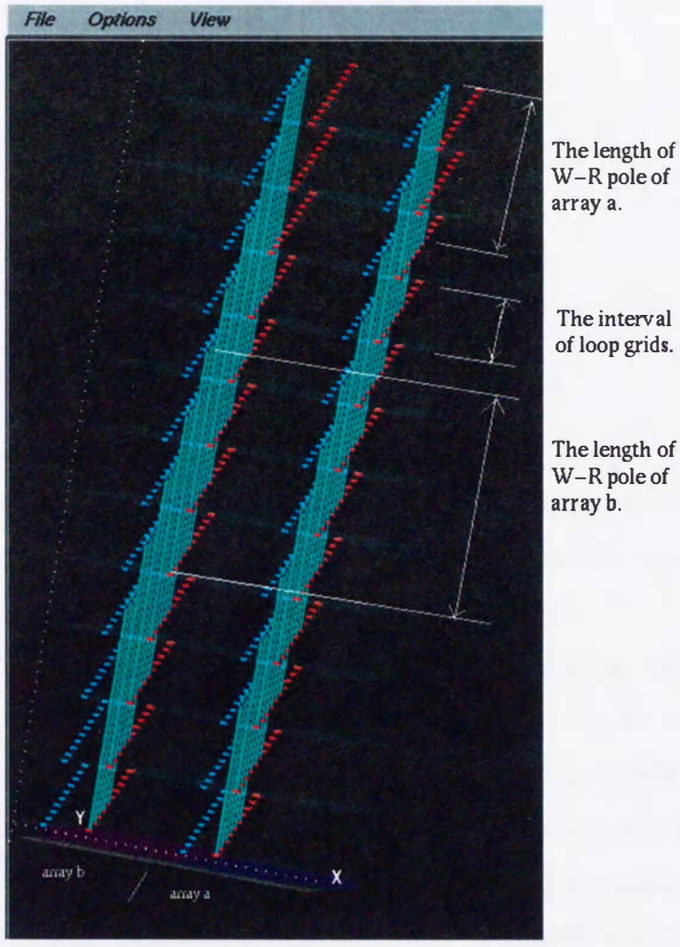


Figure 6.17: A Data Dependence View of the original example program for cycle shrinking.

```
      do 20 i = 3, 10, 2
        do 201 i = i1, i1+1
          do 18 j = 5, 10
            a(i, j) = b(i-3, j-5)
            b(i, j) = a(i-2, j-4)
18        continue
201     continue
20    continue
```

Figure 6.18: A shrinkrd program

of the program We see that the intervals of loop grids are wider than those in Figure 6.17, but the W-R poles are still longer than the intervals, which means that the loop has more potential for parallelism.

We have no general answer about when we should apply cycle shrinking and when we should not. Thus decision be based only on the difference between the length of W-R poles and the intervals of loop grids. To decide this, we need more information about such as how many times the loop will be performed and how many processors we can use in execution. It is clear, however, we cannot get parallelism more than the length of the W-R poles.

Next, we consider loop interchange in the same sample program. Loop interchange can be used when there is no difference in data dependence before and after (see section 2.1.1). DDV clearly shows whether data dependence is changed by loop interchange.

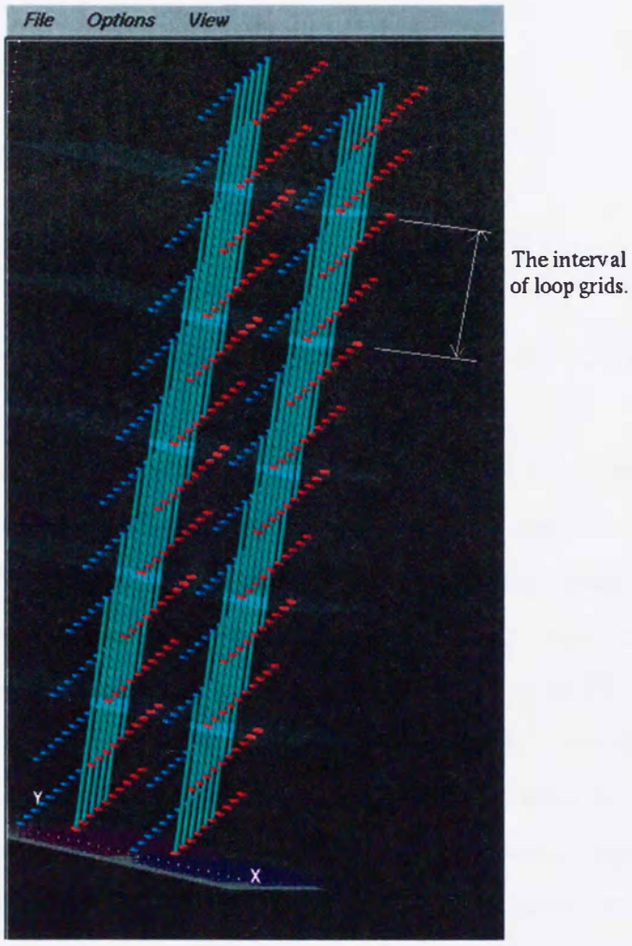


Figure 6.19: A Data Dependence View of a program after cycle shrinking.

```
      do 18 j1 = 5, 10, 4
        do 201 i = j1, j1+3
          do 20 i = 3, 10
            a(i, j) = b(i-3, j-5)
            b(i, j) = a(i-2, j-4)
18        continue
201     continue
20    continue
```

Figure 6.20: A loop interchanged and shrinked program

Figure 6.20 and 6.21 show the result of applying loop interchange to the data in Figure 6.16. The source code is shown in Figure 6.20 and DDV is shown in Figure 6.21. The data dependence of the program after loop interchange is the same as in the original because the relationship between the loop grids and poles in Figure 6.21 is the same as in Figure 6.17. But the length of W-R poles in Figure 6.21 are longer than those in Figure 6.17. Moreover, we get more parallelism from the program after loop interchange since the intervals of the loop grids in Figure 6.21 are wider than the intervals in Figure 6.17. If we apply cycle shrinking after loop interchange, we can get a program that has more parallelism than the program shown in Figure 6.19.

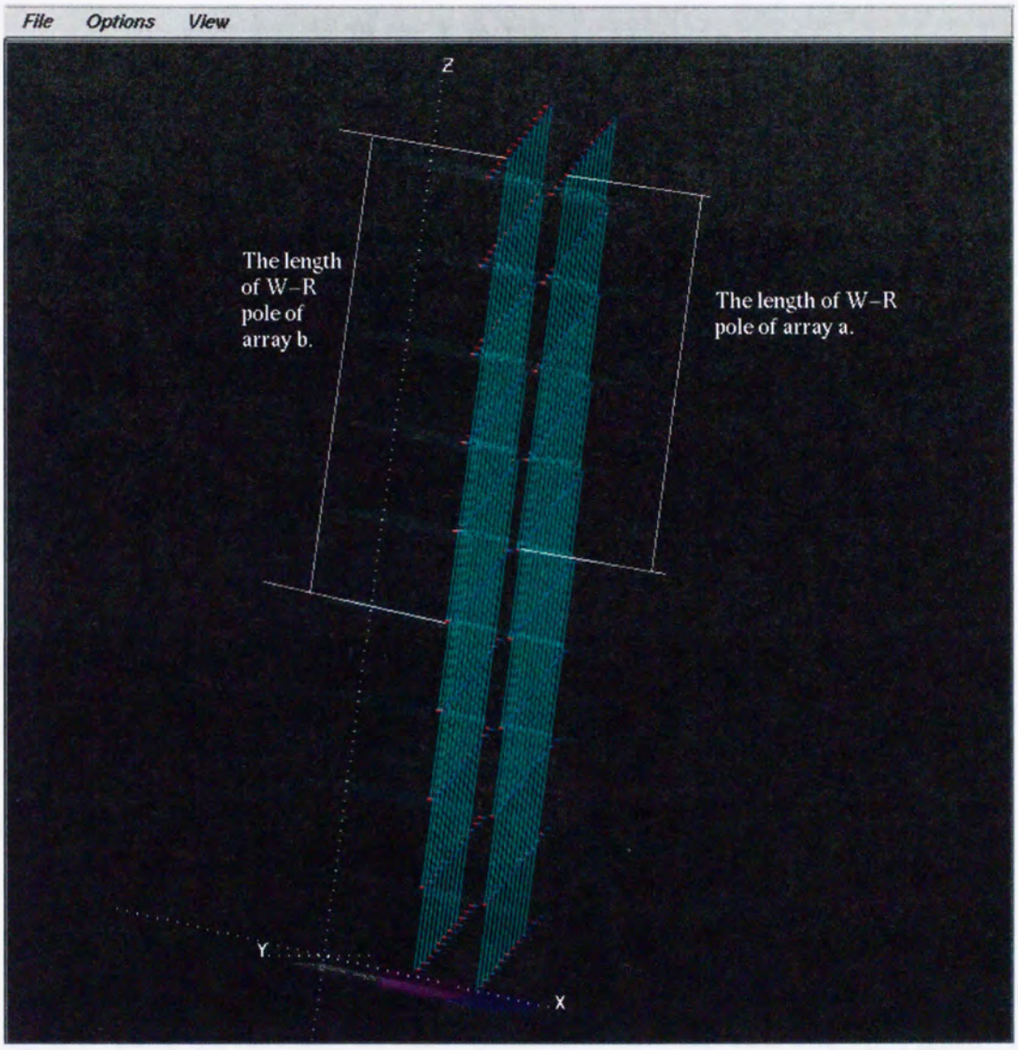


Figure 6.21: A Data Dependence View of a program with loop interchange.

```
      do 10 i = 2, n-1
       do 20 j = 2, m-1
        a(i, j) = (a(i-1, j) + a(i, j-1)
                + a(i+1, j) + a(i, j+1))/4
20     continue
10    continue
      end
```

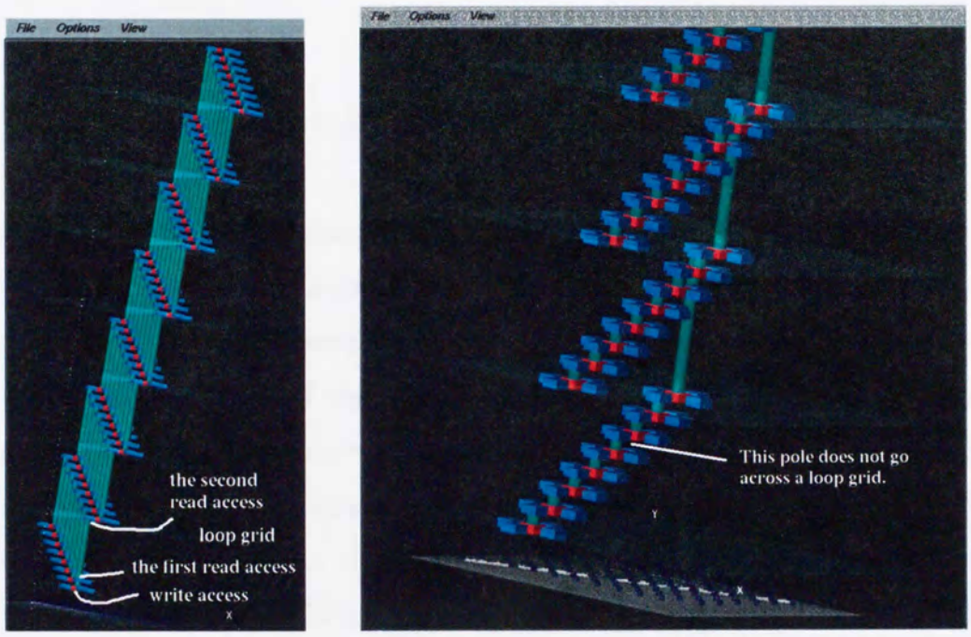
Figure 6.22: An example program of wavefront computation.

6.4.2 An example of loop skewing

Described below is an example that can be parallelized by loop skewing [2].

Figure 6.22 shows a program with a typical wavefront computation. Figure 6.23 shows the data dependences of the program, in which $n = 10$ and $m = 10$. Loop grids that correspond to the outer loop are displayed. Figure 6.23(a) shows all W-R poles and Figure 6.23(b) shows only the shortest W-R pole to each write access.

Wavefront computations have two read accesses after a write access. The first write occurs during the execution of the inner loop; the second occurs during the execution of the outer loop. After a write access to an element of array a, there are two read accesses, before and after the loop grid. In Figure 6.23(b), we find that the shortest W-R poles do not go across the loop grid. In Figure 6.23(a), however, we can see that the W-R poles go across the loop grid. Therefore, we can easily know that the loop is a wavefront computation



(a) (b)

Figure 6.23: A Data Dependence View of the program of wavefront computation: (a) with W-R poles (b) with the shortest W-R poles to each write access.

```
      do 20 j = 4, m+n-1
       do 10 i = max(2, j-m+1), min(n-1, j-2)
        a(i, j) = (a(i-1, j) + a(i+1, j) + a(i, j-1) + a(i, j+1))/4
10     continue
20    continue
```

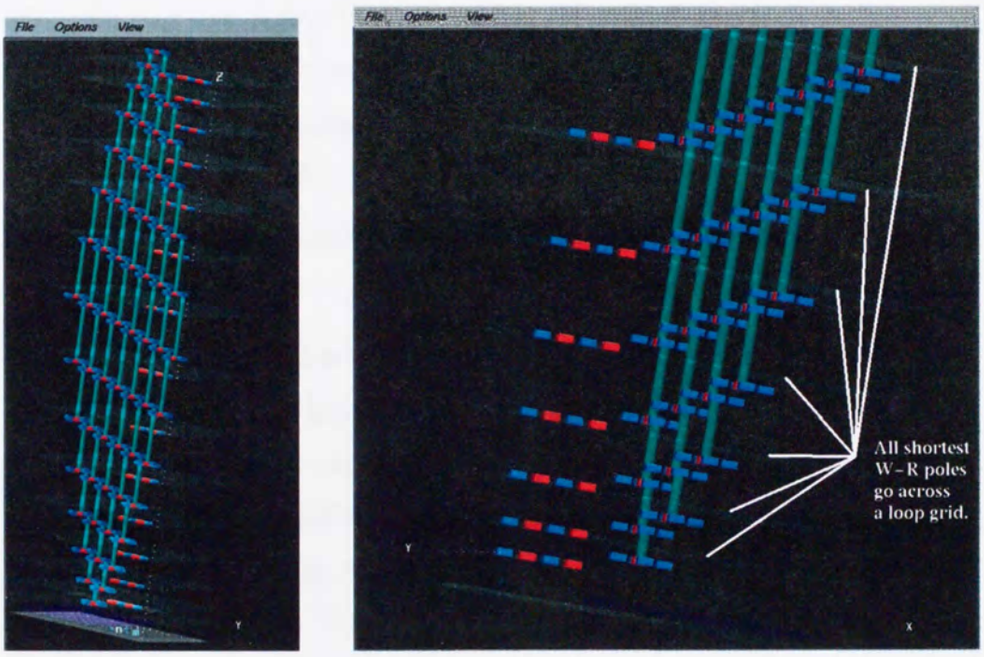
Figure 6.24: An example program of skewed wavefront computation.

and it cannot be parallelized automatically.

Figure 6.24 shows a source program that was obtained by applying loop skewing. Figure 6.25 shows the data dependences of the program. Figure 6.25 (a) shows the essence of loop skewing. Loop skewing changes the intervals of the outer loop. The intervals are no longer equal. In Figure 6.25 (b), the shortest W-R pole goes across the loop grid. It means on the outer loop all read accesses are done in different iteration from the iteration in which the write access is done. Thus, we can determine that the inner loop can be executed in parallel.

6.4.3 An example of scalar expansion

This section describes how we know to apply scalar expansion. Figure 6.7 and 6.8 show DDVs of loop 1 of Figure 6.4. The source code of this loop is represented in Figure 6.6. These figures display the data accesses when $n = 5$ and $i = 1$. Loop grids that correspond to the outer loop are displayed. The AVD map is placed at the bottom of each figure. The z-axis goes from



(a) (b)

Figure 6.25: A Data Dependence View of the program of wavefront computation with loop skewing: (a) with W-R poles (b) with the shortest W-R poles to each write access.

```
      do 1600 k = i + 1, n
       m(k) = aary(k, i) / aary(i, i)
       do 1550 j = i + 1, n + 1
        aary(k, j) = aary(k, j) - m(k) * aary(i, j)
1550   continue
1600   continue
       end
```

Figure 6.26: Scalar expansion of the program.

almost down to up, and is shown by a dotted line.

The figures tell us that the obstruction preventing this loop from being parallelized is the data dependence on variable m. This variable has been rewritten and read several times in the outer loop. These actions are obvious because of the W-R poles in Figure 6.7 and R-W poles in Figure 6.8 on m.

In this case, we can obtain a more parallelized program by changing m to an array. This is scalar expansion. Figure 6.26 shows the program after scalar expansion.

The Data Dependence View is shown in Figure 6.12. In the figure, each element of the array m (which is produced by scalar expansion) has only one write access, and read accesses to the element occur in the same region partitioned by the loop grids. It means R-W dependences are disappeared. Thus, we know that the outer loop can be executed in parallel.

6.5 Discussion

PSV is a three-dimensional graph-drawing tool that can be used to create special graphs which nodes are ordered according to some relationship: for example, program flow. We must keep the order of the relationship when we visualize the graphs. General graph-drawing tools, however, do not keep this order [29]. Instead, they follow other criteria such as reducing crosses of arcs. Therefore, CFG and HTG can be visualized with general graph-drawing tools, but the figures produced cannot be understood intuitively.

One of the general problems of visualization is how we treat large amounts of data (Figure 6.15). Although 350 lines of code may not seem large, a program that has more than 1000 lines usually consists of several modules (subroutines). An HTG is made for a module, so PSV is not usually required to visualize large amounts of date. But we think a more serious problem is that we have no way to know the relationship between modules. For example, Figure 6.14 shows a module that includes many function calls , but we cannot indicate the program structures of the invoked functions. We should consider a PSV-PSV connection for function calls. It is simple expansion and not so difficult.

DDV can suggest whether to apply loop reconstruction methods (see chapter 6). DDV is especially useful for loops that have complicated index expressions. But of course DDV is not all powerful, and does not always specify that loop reconstruction methods should apply. For instance, finding a part to which loop distribution should be applied is difficult because loop distribution is a loop reconstruction method that parallelizes a sentence

rather than a loop. DDV is designed to let users show a loop or loop iterations rather than a sentence, so DDV is not so useful in loop distribution.

We can also have difficulty finding the best loop reconstruction methods when there are many variables and various data dependences in a loop. We believe a compiler can suggest which parts of dependence disturb parallelization. We can then visualize the suggestion.

Users should know that DDV visualizes the data dependence of an execution through the simulation module of NaraView. The simulation module executes only a part of program and the parts of iterations that are needed to visualize the data dependence of a specified loop. Therefore, there is no guarantee that there is no data dependence when we cannot find dependence poles in a figure.

Although we do not discuss them in this thesis, we think DDV has various other uses. For example, we can investigate how data should be partitioned and distributed by mapping data to the appropriate layout on the x-y plane. DDV is not almighty on visualizing data dependence, but it is helpful in understanding the program and debugging.

Chapter 7

Conclusion

This thesis described NaraView, a program visualization system for parallelizing compilers, which plays an important part of an interactive compilation environment for parallelization in the phase of compilation. Program visualization reorganizes a program according to some models and visualizes it.

In chapter 3, we showed the requirements for NaraView and the architecture of NaraView. The requirements for NaraView are to let users easily understand information for parallelization from a compiler. NaraView visualizes the analysis of a program extracted by a parallelizing compiler by four views. We described in detail two views of NaraView: the Program Structure View and the Data Dependence View in chapter 4 and 5.

In chapter 4 we described the Program Structure View (PSV). It shows the user three-dimensional visible objects that visualize the structure of given programs for intuitive understanding. PSV visualizes each sentence of a program as a colored cube and puts it in three-dimensional space. The three axes of the space means program flows, hierarchical levels of loop structure and

measure of parallelism. PSV allows users to investigate each loop that does not seem to have parallelism at a glance with the original source program.

In chapter 5, we represented the Data Dependence View (DDV) that shows the data dependence in the loop focused on by using the Program Structure View. We proposed variable-oriented data dependence model as a basic model of DDV. DDV displays each access as a colored cube and each data dependence as a colored pole. DDV also shows loop grids to indicate the beginning of each iteration of a loop or specified iterations. Therefore users can read patterns of data dependence from the relationship between loop grids and poles. This view is useful for comparing and selecting transformation methods to parallelize given programs.

Chapter 6 includes several examples of PSV and DDV. The examples show PSV, DDV and a collaboration of the views, which are useful for non-expert users for parallelization. In section 6.2, we explained the typical way to use NaraView. And in section 6.3 and 6.4, we showed how we can interpret PSVs and DDVs respectively.

Many researchers have claimed the usefulness of interactive compilation environments. We especially have claimed and shown the usefulness of program visualization in an interactive compilation environment.

In future work, we will be able to extend NaraView to allow users to modify a given source program using a visual interface. For example, when some redundant dependences are detected in a view, users can remove the dependences by cutting the dependence poles without modifying the source program. If this operation succeeds in finding parallelism, the source program is updated by NaraView automatically.

In this thesis, we assume we have infinite processors and infinite memories. However, in practical, we run a parallel program on finite number of processors and finite memories. Therefore, we should extend our system for finite number of processors and memories for practical use.

To give users more assistance on parallelizing a program, as we discussed in section 6.5, we may specify the obstacle of parallelization and consider giving users suggestions about which parallelization methods can be applied. We have implemented a causal explanation system with an abduction procedure for failure in applying loop transformation [25]. The system informs the user the obstacle of parallelization and explains whether a specified loop transformation method can be applied on a specified loop by the abduction procedure.

Acknowledgement

I would like to express my appreciation to Professor Keijiro Araki of Kyushu University for the supervision and encouragement to complete this thesis.

I also wish to express my thanks to Professor Makoto Amamiya and Professor Hiroto Yasuura of Kyushu University for the helpful advice.

I wish to thank Professor Susumu Yamasaki of Okayama University. He understands my research very well and keep watching it warmly.

I would like to thank all members related to NaraView project. Professor Kazuki Joe of Nara Women's University has discussed the basic ideas of NaraView with me frequently. Doctor Tsuneo Nakanishi of NAIST gave his knowledge of parallelizing compilers to me. Ms. Satoko Kiwada implemented many parts of NaraView.

I also wish to thank Professor Constantine D. Polychronopoulos of Illinois University and Professor Yoshitoshi Kunieda of Wakayama University for their useful advice.

I would like to give thanks to Ms. Julie Yamamoto for the advice about improving my English.

Last, I would like to express my appreciation to Doctor Kenzo Iwama. He gave me the chance to become a researcher.

Bibliography

[1] Aho, A. V., Sethi, R. and Ullman, J. D. "Compilers: Principles, Techniques and Tools", Addison-Wesley, Reading, Massachusetts, 1986.

[2] Bacon, D. F., Graham, S. L. and Sharp, O. J. "Compiler transformations for high-performance computing", ACM Computing Survey, vol. 26, no. 4, pp. 345-420, 1994.

[3] Banerjee U. "Dependence Analysis for Supercomputing", Kluwer Academic Publishers, 1988.

[4] Banerjee, U. "Loop Transformations for Restructuring Compilers: the foundations", Kluwer Academic Publishers, 1993.

[5] Browne, J. C., Hyder, S. I., Dongara, J., Moore, K. and Newton, P. "Visual programming and debugging for parallel computing", IEEE Parallel & Distributed Technology, vol. 3 no. 1 pp. 75-83, 1995.

[6] Chuah, M. C. and Eick, S. G. "Managing software with new visual representations", Proceedings of Information Visualization '97 pp. 30-37, 1997.

[7] Cypher, A. ed. "Watch What I Do: Programming by Demonstration", MIT Press, 1993.

[8] Frumkin, M., Hribar, M., Jin, H., Waheed, A. and Yan, J. "A comparison of automatic paralllization tools/compilers on the SGI Origin 2000", Proceedings of the 1998 ACM/IEEE SC98 Conference (CD-ROM), 1998.

[9] Girkar, M.B. and Polychronopoulus, C. D. "The hierarchical task graph as a universal intermediate representation", International Journal of Parallel Programming, Vol. 22, No. 5, pp. 519-551, 1994.

[10] Hall, M.W., Harvey, T.J., Kennedy, K., McIntosh, N., McKinley, K.S., Oldham, J.D., Paleczny M.H. and Roth, G. "Experiences using the ParaScope Editor: an interactive parallel programming tool", SIGPLAN Notice, vol. 28, no. 7, pp. 33-43, 1993.

[11] Heath, M. T. and Etheridge, J. A. "Visualizing the performance of parallel programs", IEEE SOFTWARE vol. 8, no. 5, pp. 29-39, 1991.

[12] Ierotheou, C.S., Johnson, S.P., Cross, M. and Leggett, P.F. "Computer aided parallelization tools(CAPTools) – conceptual overview and performance on the parallelisation of structured meshcodes" Parallel Computing (Netherlands), vol. 22, no. 2, pp. 163-95, 1996.

[13] Iwasawa, K., Kurosawa, T., Kikuchi, S. "Parallelization method of Fortran DO loops by parallelizing assist system", Transactions of Information Processing Society of Japan, vol. 36, no. 8, pp. 1995-2006, 1995. (in Japanese)

[14] Kennedy, K., McKinley, K. S. and Tseng C.-W. "Interactive parallel programming using the ParaScope Editor", IEEE Transactions on Parallel and Distributed systems, vol. 2, no. 3, pp. 329-341, 1991.

[15] Koike, H. and Aida, M. "A bottom-up approach for visualizing program behavior", 11th IEEE Symposium on Visual Languages, pp. 91-98, 1995.

[16] Koike, H. and Chu, H.C. "VRCS: Integrating version control and module management using interactive three-dimensional graphics", Proceedings of IEEE Symposium on Visual Languages 97, pp. 170-175, 1997.

[17] Kraemer, E. and Stasko, J.T. "The visualization of parallel systems: an overview", Journal of Parallel and Distributed Computing, vol. 18, no. 2, pp. 105-17, 1993.

[18] Liao, S.-W., Bosch Jr., R.P., Ghuloum, A. and Lam. M.S. "SUIF Explorer: A programming assistant for parallel machines", Proceedings of the Second SUIF Compiler Workshop, August 1997. http://www-suif.stanford.edu/~ sliao/papers.html

[19] Marriot, K. and Meyer, B. "Visual Language Theory", Springer, 1998.

[20] Munzner, T. "H3: Laying out large directed graphs in 3D hyperbolic space" Information Visualization '97, pp. 2-10, 1997.

[21] Novack, S. and Nicolau, A. "VISTA: The visual interface for scheduling transformations and analysis", Languages and Compilers for Parallel Computing. 6th International Workshop Proceedings, pp. 449-460, xi+655, 1993.

[22] Polychronopoulos, C.D. "Parallel Programming and Compilers", Kluwer Academic Press, 1988.

[23] Polychronopoulos, C.D., Girkar, M. B., Haghighat, M. R., Lee, C. L., Leung, B. P. and Schouten, D. A. "Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors", Proceedings of the 1989 International Conference on Parallel Processing, volum II, pp. 39-48, 1989.

[24] Reed, D. A., Shields, K. A., Scullin, W. H., Tavera, L. F. and Elford,C. L. "Virtual reality and parallel systems performance analysis", IEEE Computer vol. 28, no. 11, pp. 57-67, 1995.

[25] Sasakura, M. "A causal explanation system with abduction procedure for failure in applying loop transformation", Journal of Japanese Society for Artificial Intelligence, vol. 14, no. 5, (in press) 1999. (in Japanese)

[26] Shu, N.-C. "Visual Programming", New York: Van Nostrand Reinhold Company, 1988.

[27] Simmons, M.L., Hayes, A. H., Brown, J.S. and Reed, D. A. ed. "Debugging and Performance Tuning for Parallel Computing Systems", IEEE Computer Society Press, 1996.

[28] Stasko, J., Domingue, J., Brown, M.H. and Price, B.A. ed. "Software Visualization", MIT Press, 1998.

[29] Sugiyama, K. and Misue, K. "Visualization of Structural Information: Automatic Drawing of Compound Digraphs", IEEE Transaction on Sys-

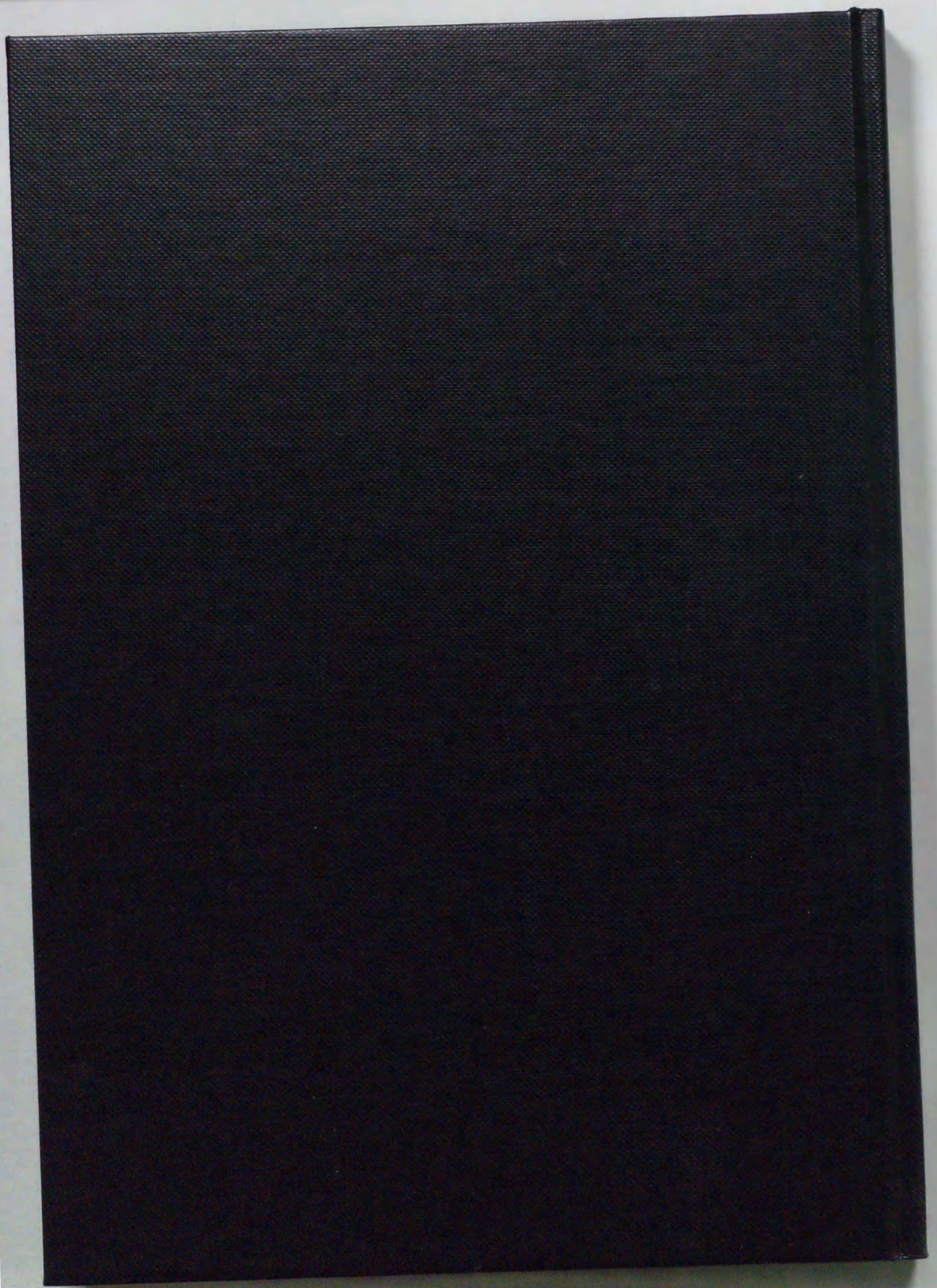
tems, Man and Cybernetics, Vol.21, No.4, pp.876-892, July/August 1991.

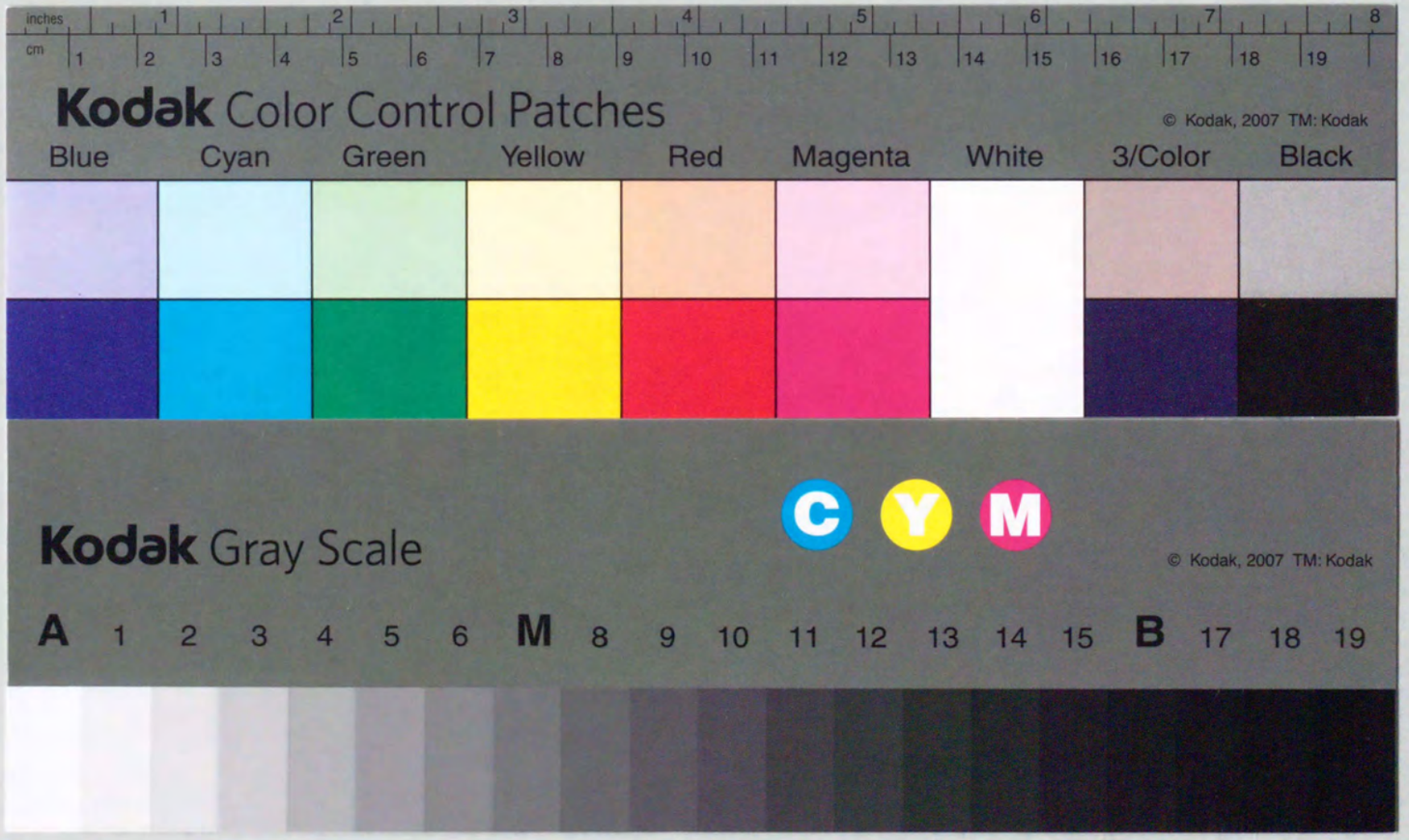
[30] Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S-W., Tseng, C-W., Hall. M.W., Lam, M.S. and Hennesy, J.L. "SUIF: An infrastructure for research on parallelizing and optimaizing compilers", ACM SIGPLAN Notices vol. 29, no. 12, pp. 31-37, 1994.

[31] Wolf, M. E. and Lam, M. S. "A Loop transformation theory and an algorithm to maximize parallelism", IEEE Transactions on Parallel and distributed systems, vol. 2, no. 4, pp. 452-471, 1991.

[32] H. Zima and B. Chapman "Supercompilers for Parallel and Vectoer Computers", Addison-Wesley, 1991.

Index




inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19