Source-level Debugging of Automatically Parallelized Programs

Robert Cohn

23 October 1992
CMU-CS-92-204

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Supported in part by an Intel Corporation Graduate Fellowship and in part by the Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing," ARPA Order No. 7330. Work furnished in connection with the research is provided under prime contract MDA972-90-C-0035 issued by DARPA/CMO to Carnegie Mellon University and under its subcontract No. 334918-58792 with Networks Systems Corporation.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Intel, DARPA or the U.S. government.

Keywords: Debugger, compiler, parallelism, distributed memory, optimization

School of Computer Science

Accesion For	
NTIS CRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By Per Ltr.	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

DOCTORAL THESIS
in the field of
Computer Science

Source-level Debugging of Automatically Parallelized Programs

ROBERT S. COHN

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

Thomas Gross — THESIS COMMITTEE CHAIR — 10/27/92 — DATE

A. T. Kung — THESIS COMMITTEE CHAIR — 10/27/92 — DATE

H. Thomas — DEPARTMENT HEAD — 10/29/92 — DATE

APPROVED:

R. Rashid — DEAN — 10/29/92 — DATE

Source-Level Debugging of Automatically Parallelized Programs

Robert Cohn

Abstract

Parallelizing compilers automatically translate a sequential program into a parallel program. They simplify parallel programming by freeing the user from the need to consider the details of the parallel architecture and the parallel decomposition. A source-level debugger for automatically parallelized programs hides the parallelism from the user by providing the illusion that the original sequential program is executing.

To provide a source-level view, a debugger has two tasks. The first task is to make it appear that programs execute operations in source order. To exploit parallelism, compilers relax the ordering constraints implied by the semantics of the source language. As a result, operations are not executed in source order and when a user is debugging a program, it may appear that variables are updated out of order. Sequential languages and languages with explicit parallelism share this problem. Dynamic order restoration is a method for making it appear that operations are executed in source order.

The second task is to hide the decomposition of data and computation. A parallelizing compiler partitions data, duplicates variables, and changes the structure of loops. To provide a source-level view, the debugger must be able to map variables and statements in the source programs to their corresponding variables and statements in the target program. We call this structural mapping.

This thesis describes a method for implementing dynamic order restoration and structural mapping in a debugger for automatically parallelized programs. Examples are taken from the domain of loop-based parallelism.

Acknowledgments

I would like to thank my advisors Thomas Gross and H. T. Kung. Without Thomas' support and technical feedback I could not have finished this thesis. I am grateful to Kung for asking the tough questions that kept my work honest.

I would also like to thank the members of my thesis committee. My discussions with Bernd Bruegge improved the technical content and presentation of the thesis and David Padua gave me a valuable outside perspective on my work.

I would like to thank my friends at CMU for making my time here so enjoyable. Their friendship and support, especially in the final months of thesis was a great help. Special thanks go to Michael Hemy, who carefully read my thesis.

I would like to thank my brother Richard, who brought me to CMU originally, and also my parents and sister.

Table of Contents

CHAPTER 1 Introduction

Programming parallel machines is much more difficult than programming sequential machines. Writing a parallel program requires that the programmer divide the computation among a set of processors so that the load is equally balanced, add synchronization if the target architecture is a shared memory machine, and add communication if the target is a distributed memory machine. Writing a program that is optimized for a particular architecture makes these tasks especially difficult.

Parallel program generators, also called parallelizing compilers, can ease the programmer's burden by automating some or all of this work. Some compilers provide a programming model that allows the programmer to express the algorithm in a manner that is independent of the number of processes; the compiler decides how to best distribute the computation. For distributed memory machines, compilers can provide a shared memory programming model and automatically distribute data.

Compilers may make writing programs easier but do not solve the whole problem—coding is just one step in creating a new program. In particular, debugging can be a very time consuming and tedious process. Debugging an automatically parallelized program is difficult because the user did not write the target program that actually executes. Understanding how the program works would require that the user know all the details of the parallelization that the compiler was trying to hide.

My thesis is that a debugger for automatically generated parallel programs can provide a sequential view of parallel execution. In other words, parallelizing compilers can have source-level debuggers [Pineo 91][Cohn 91]. The user can set breakpoints and inspect variables as if they are debugging their source program, even though a parallelized program is executing.

The rest of the chapter is organized as follows. In Section 1.1, we illustrate some of the issues in source-level debugging with an example. A description of the various programming and machine models employed for parallel machines and their impact on debugging can be found in Section 1.2. Section 1.3 introduces the approach of this thesis for building debuggers. Section 1.4 discusses some related work, and Section 1.5 outlines the presentation of the thesis.

1.1 Debugging issues: an example debugging session

There is a wide spectrum of programming models and machine architectures. Programming models and architectures range from sequential to MIMD parallel and from shared to distributed memory. Each combination of programming model and architecture has different requirements for source-level debugging. To introduce the main issues in implementing source-level debugging, we will use a simple example where the programming model is sequential and the actual machine is a MIMD, distributed memory machine.

FIGURE 1-1 depicts a sequential program that initializes all the elements of the array `A` to 1 and a two processor parallel program generated by a compiler. In the parallel program, processor 0 executes all the even iterations of the original loop and processor 1 executes all the odd iterations. On a distributed memory machine, the data must be partitioned. For this program, processor 0 has all the elements of `A` with even indexes and processor 1 has all the elements of `A` with odd indexes.

The main issues that we address in this thesis are setting breakpoints, examining and modifying variables, and reporting the current location of the program counter in the program. We use the above program to give examples of the problems associated with these issues. Assume that the user starts execution of the program and after some time, the interrupts the program. As depicted in FIGURE 1-2, processor 0 is executing the loop control of the `for` statement and processor 1 is executing the body of the loop. In the figure, the stop signs indicate the source lines that each processor will execute next. The debugger must report one statement in the source program as being the current location. To determine this, the debugger must know the relationship between lines in the source program and lines in the parallel program. In this example, the correspondence is simple, but it can be complicated when the compiler applies transformations that insert and delete lines and restructure the program.

Next, the user tries to inspect the value of array element `A[2]`. The array `A` has been divided into two smaller arrays and, as shown in FIGURE 1-3. Array element `A[2]` resides on processor 0 and is called `A0[1]`; the debugger must know the relationship between data in the source program and data in the target program. This relationship is not necessarily static. For example, each processor has its own copy of the loop counter `i` and choosing which copy is the correct one to display is dependent on the state of the program.

Before actually displaying the value, the debugger must be sure that the value that is currently in memory is the same one the user would have seen if the source program were executing. If processor 0 has executed one iteration of its loop, and processor 1 has executed three iterations, then the memory of each processor would appear as is shown

FIGURE 1-1 Sequential program and its parallel version

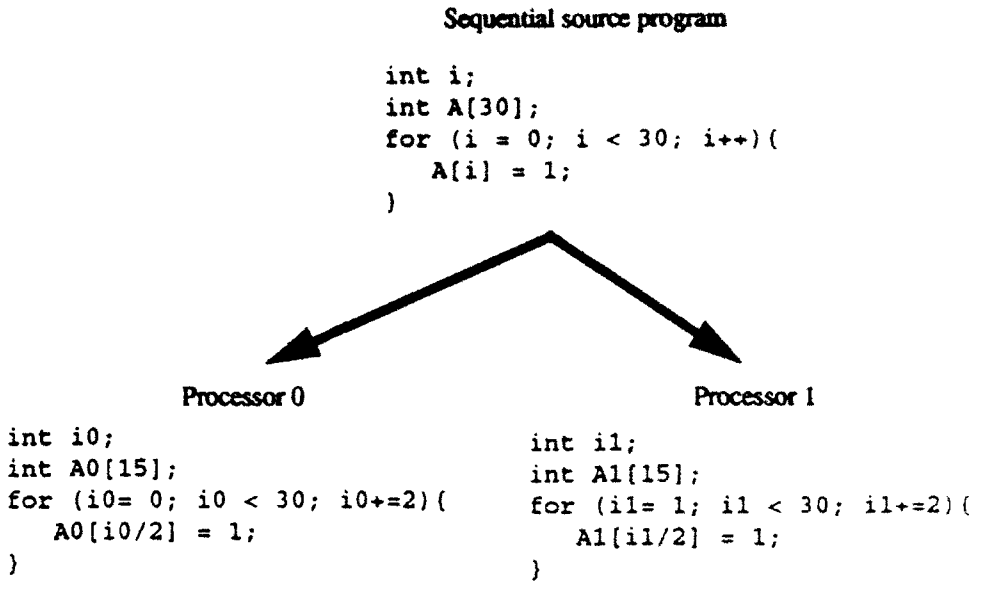
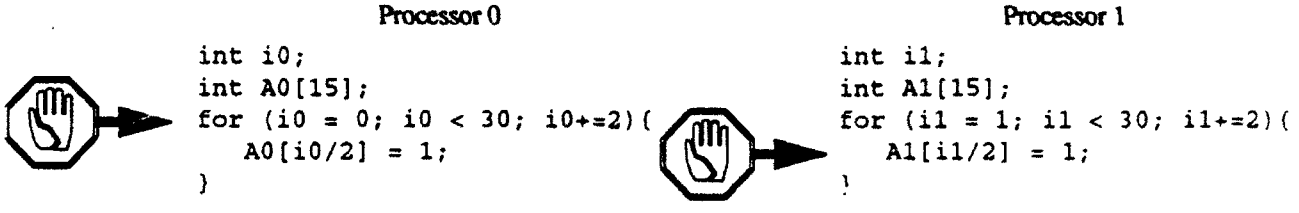
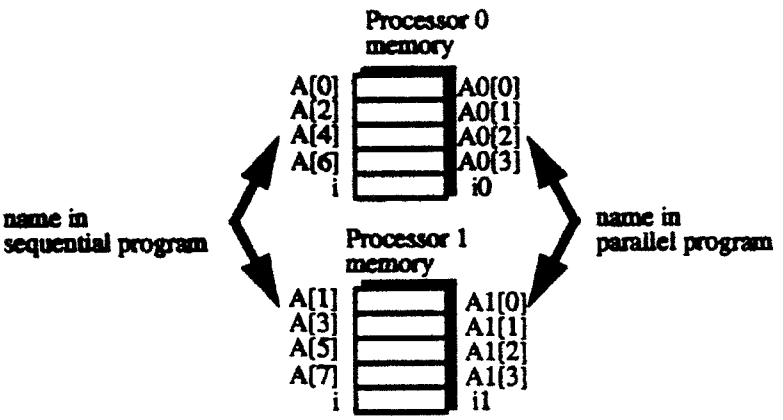


FIGURE 1-2 When the user aborts the program, processor 0 is stopped at the `for` statement and processor 1 is executing the assignment.



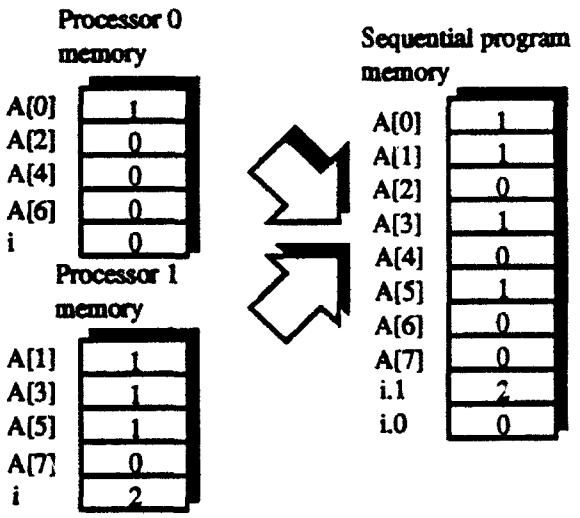
in FIGURE 1-4. If we merged both memories to produce the memory of the source program, then it would contain the values as shown in the right side of the figure. This state clearly cannot be the result of execution of the sequential program. The loop in the source program walks through the array `A` sequentially, but array element `A[3]` has been initialized while element `A[2]` has not. The debugger must detect when such an inconsistency has occurred so that it does not give the user any misleading information.

FIGURE 1-3 The relationship of names of variables in the source and target programs



In this case, telling the user that `A[2]` has a value of 0 and `A[3]` has a value of 1 is misleading because the programmer might incorrectly conclude that the program failed to initialize element `A[2]`. In this situation, one possible response for the debugger could be that `A[2]` has a value of 0 and that `A[3]` and `A[5]` cannot be examined.

FIGURE 1-4 The state of memory after processor 0 has executed 1 iteration and processor 1 has executed 3 iterations



Setting breakpoints can have similar problems. In our example, iteration 3 of the source program has already executed, but iteration 2 has not yet executed. If the user sets a

breakpoint in the loop, the program reaches the breakpoint for iterations 2 and 4 has already run past the breakpoint for iteration 3. In this situation, the debugger should warn the user that some breakpoints might be skipped. When breakpoints occur, the debugger should present them to the user in the order that they would have occurred in the sequential program, even though they may occur out of source order.

We classify the problems described above into two categories, *dynamic order restoration* and *structural mapping*. These are the fundamental services that a debugger for parallelized program must provide. This thesis describes how to implement these services.

Dynamic order restoration ensures that the *observable* order of execution of operations is the same as in the source program. If the semantics of the source language imply that operation A is executed before operation B, and in the target program operation A can be executed before or after operation B, then a debugger that does dynamic order restoration behaves as if operation A is always executed before B.

Parallelizing compilers relax the ordering constraints implied by the semantics of the source language to increase parallelism. The semantics of a sequential program imply that only one operation at a time is executed; a compiler converts a sequential program into a parallel program where more than one operation can be executed simultaneously Compilers remove ordering constraints even when the source language has explicit parallelism. We explain this in more detail in Section 1.2.

If operations are executed out of source order, then as described in the example of FIGURE 1-4, variables may be updated out of order. A debugger that does dynamic order restoration must make it appear that they are updated in order. This may include preventing the user from examining variables at some points in the program. Another possible result of out of order execution is that breakpoints may not occur in source order. The debugger must present the breakpoints to the user in source order.

Structural mapping relates the source lines and variable names in the source and target programs. In the previous example, determining that array element `A[3]` resides in the memory of processor 1 and that it is called `A1[1]` is structural mapping.

Structural mapping is also necessary in debuggers for sequential machines; relating source code and machine code is an example. In addition, programs can be restructured by optimizing compilers. Dynamic order restoration is unique to debuggers for parallelizing compilers and requires new mechanisms; the debugger must be able to determine the order in which operations executed. The nature of dynamically reordered operations also provides for flexibility that is not present for sequential programs. If a constraint is removed and two operations can occur in order or out of order, then the debugger can constrain execution so that they occur in order.

1.2 The semantic gap between programming models and architectures

The difficulty of source-level debugging depends upon the gap between the programming model and the code that executes on the parallel machine. This is in turn depen-

dent on the semantics of the source language, the architecture of the parallel machine, and the way the compiler implements the source language on the machine. All compilers create the need for structural mapping. Some, but not all, need dynamic order restoration. In this section, we sketch out the spectrum of programming models provided by parallelizing compilers and discuss what problems are introduced when they are implemented on different architectures.

The programming model that the source language allows can be very different from the architecture. From a debugging standpoint, the two most important components of the model are the user view of processors and memory.

In a programming model with a low level of abstraction, the user must explicitly write a program for each processor in the system. Compilers can lift the level of abstraction by adding constructs that directly support parallelism. The `cobegin/coend` and `doall` are two examples. The semantics of these constructs is that there is only one thread at the beginning of the construct, one thread is spawned to execute each clause in the `cobegin/coend` or each iteration of the `doall`. After all the threads have completed their assigned work, there is a join and only one thread continues execution. Data parallel operations are similar; there is one thread of control before the operation begins, the data parallel operation is executed in parallel, and there is one thread after the operation completes. These constructs hide the architecture of the parallel machine and the details of the mapping. The user does not need to know how many processors there are, whether the architecture is SIMD or MIMD, and does not need to decide which operations are executed on each processor. The programming model can be moved one step farther from the architecture by providing a sequential model of computation. In this case, the user does not specify the parallelism—the compiler automatically detects it.

The user view of memory in programming models ranges from distributed memory, where a processor may only access data in its local memory, to shared memory, where a processor may access all data uniformly. For programming models with distributed memory, data can only be moved between processors when the originating processor sends the data and the destination processor receives it. Other systems provide programming models that are a hybrid; data can be fetched from another processor's memory without the owner sending it, or data can be put in another processors's memory without the destination processor receiving it.

A semantic gap exists when the programming model doesn't match the architecture. The rest of this section explains which combinations of programming models and architectures require structural mapping and dynamic order restoration. If the compiler provides a shared memory model on a distributed memory machine, then the debugger must also provide the illusion that there is a single address space by translating the variable names and array subscripts in the source program to names and subscripts in the parallel program.

There is a close match between a SIMD architecture and languages with data parallel operations; the basic operations of the machine are element-wise. Executing a sequential language or a language with parallel loops on a SIMD machine requires using the vectorization techniques of strip mining, loop distribution, etc. These techniques introduce some structural changes but since the target architecture has a single thread of control, no dynamic order restoration is required.

A MIMD architecture creates the need for order restoration. If a programming model with explicit processes is implemented on a MIMD machine, then there is little difference between the programming model and architecture. However, if a restricted parallel model, such as parallel loops[Poly 89][Balasundaram 89][Mehrotra 90][Sussman 91] or data parallel model[Chatterjee 91][Fortrand 92][Knobe 90] is implemented on a MIMD machine, then there is a potential for dynamic reordering to occur, depending on the implementation. A straightforward, but inefficient way to implement a parallel loop is to spawn off processes at the beginning of the loop, have each loop execute some of the iterations, and then kill all but a master process after the loop terminates. If a parallel loop is implemented in this manner, the semantics of the source language and the actual execution are a close match. However, most compilers use a more efficient method for implementing a parallel loop. All the processes execute the sequential code outside the parallel loop, each process executes some of the iterations of the loop, and all processes continue execution after the loop terminates[Booth 86][Cytron 90][Chatterjee 91][Tseng 89]. The processes do not necessarily synchronize at the beginning or end of the loop, unless data dependences require it. This implementation is more efficient because the program does not need to spawn and kill processes, and the lack of barrier synchronizations can reduce idle time. With such an implementation, it is possible for one processor to be executing code before the beginning of the loop while another processor is executing iterations of the loop. If the program were to stop in that state, the order of execution would violate the semantics of the language, which requires dynamic order restoration.

Executing data parallel operations on a MIMD machine is similar to executing parallel loops; if a barrier synchronization is done before and after every data parallel operation, then the semantics of the language and the actual execution are a close match. If the synchronization is not needed, then it can be removed to make the program more efficient. When the synchronization is removed, two processors can be executing different operations at the same time[Chatterjee 91], which violates the semantics and creates a need for dynamic order restoration in the debugger.

To summarize, structural mapping is always needed. The debugger must associate statements in the source and target programs. If the compiler provides a shared memory model on a distributed memory machine, then the debugger must also provide that model. When the target program does not obey the ordering constraints of the source program, as is the case for data parallel, parallel loops, and sequential models implemented on MIMD machines, then the debugger must also provide dynamic order restoration.

The biggest gap between source program and execution occurs when a single address space, sequential program is executed on a distributed memory MIMD machine. This is the main focus of our thesis. However, the methods explained in the thesis apply even when the programming model is not sequential with a single address space.

1.3 Overview of the approach

The principal insight of our approach is that the structural aspects of parallelization can be expressed in the context of a program which has *not* been parallelized. For example, in FIGURE 1-5 we have rewritten the sequential program in FIGURE 1-1 as another

sequential program where the distribution of A into two smaller arrays and the duplication of the loop counter i is exposed but the parallelism is not visible.

FIGURE 1-5 A restructured version of the sequential program in FIGURE 1-1.

```
int i0,i1;
int A0[15],A1[15];
for (i0 = 0,i1 = 1; i0 < 30; i0 +=2,i1+=2){
   A0[i0/2] = 1;
   A1[i1/2] = 1;
}
```

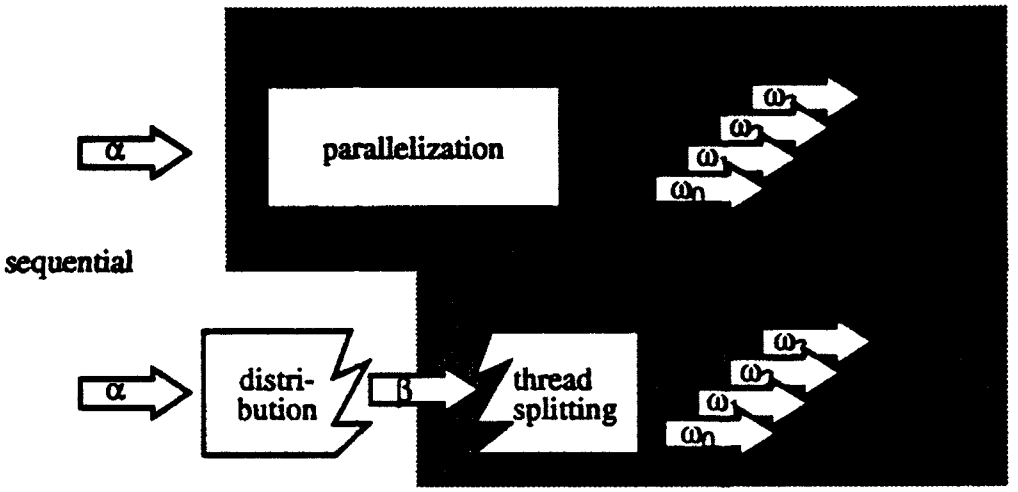
Our methodology for constructing debuggers separates the dynamic ordering and structural mapping issues by breaking the parallelization transformation of compilation into two phases. Instead of translating the program directly from sequential to parallel, as is depicted in the top of FIGURE 1-6, the compiler uses the 2 phase process shown in the bottom of FIGURE 1-6. The first phase is called distribution and exposes the restructuring necessary for parallelization. It is a sequential to sequential program transformation. and its output is the β program. The second phase is a parallelization transformation called *thread splitting* and its output is the parallel ω program. Thread splitting is a simple and general transformation which extracts multiple threads of control from a single one. A debugger is constructed by building a debugger for distribution and another debugger for thread splitting. The debugger for the original transformation can then be built by composing the two debuggers. The source-level for the debugger for the distribution transformation is the α program and the source-level for the thread splitting debugger is the β program.

Structural mapping is only done is the debugger for distribution. Dynamic order restoration is only done in the debugger for thread splitting. To implement a different parallelizing transformation, we change the distribution, but thread splitting is the same. Since thread splitting is the only transformation that introduces parallelization in the system, its debugger can be reused for all parallelizing compilers. In effect, the part of the debugger that manages parallelism is isolated and is independent of the parallelizing transformations that a compiler uses.

1.4 Related work

The two main features that are present in parallel program debugging but are not in sequential program debugging are nondeterminism and the extra information related to multiple threads of control. Nondeterminism makes debugging difficult because incorrect behavior might not be reproducible. Multiple threads of control make debugging more difficult because the user must filter through extra information when debugging (e.g. breakpoints can occur on more than one processor at the same time).

FIGURE 1-6 Compiler transformation from sequential to parallel



To solve the first problem, researchers have worked on systems to detect nondeterminism when it is an undesirable property of a program and to control it when it is necessary. Detecting nondeterminism usually relies on a combination of static analysis of programs and run-time checking [Netzer 91][Callahan 90][Emrath 89][Wang 90][Dinning 91]. These tools are intended for shared memory machines and check for unsynchronized accesses to shared variables. In our work, we assume that the program has been automatically parallelized, so nondeterminism of the type detected by these debugging tools cannot occur if the compiler is correct. It is possible however, that the user can mistakenly direct the compiler to parallelize a loop that cannot be correctly executed in parallel, which could create a nondeterministic program. Nothing prevents the user from employing other tools in conjunction with a source-level debugger in this case. Once all synchronization bugs have been removed, source-level debugging can be used.

If a program is intentionally nondeterministic, then debugging is difficult because the problem may not be reproducible when using the debugger. To solve this problem, researchers are studying how to make the behavior reproducible, either by restricting execution so that it is reproducible, debugging from traces of a single run, or a combination of both[Forin 88] [Miller 88][Leblanc 87][Tolmach 91][Bacon 91]. With a MIMD execution model, many different possible interleavings of instruction executions are possible even for a program with deterministic results. This can be viewed as a type of nondeterminism since the interleaving varies from execution to execution. By providing a source view of execution, we are in effect choosing a single interleaving as the correct one. In this respect our work is similar to Tolmach [Tolmach 91], who makes a program deterministic by imposing a total ordering on access to *shared* objects, but optimistically allows the program to run in parallel. Roll back, or restoring the program to a previous state, is used when the optimistic parallel execution violates the chosen

ordering. Our work differs because we give the appearance of a total ordering of access to *all* objects in the program but do not necessarily restrict execution to the total ordering. If Tolmach's method were used to impose a total ordering for all objects, there could not be any parallelism during execution.

The second feature that distinguishes parallel program debugging from sequential debugging is the extra information related to having multiple threads of control. For example, a sequential machine can only have one breakpoint at a time, but on a parallel machine, every processor can hit a different breakpoint at the same time. Parallel machines can also generate more information in the same time than sequential machines. To solve this problem, researchers have studied ways to filter and present the information that is presented to the user. Visualization and auralization have been used to graphically present information about the status of processors[Heath 91] [Bailey 88][Zernik 91][Francioni 91]. Another way to reduce the amount of information that a user must see is to test for high level events associated with the behavior of the program rather than low level events such as source line breakpoints. Researchers have studied how to detect and report events promptly without disturbing the state of the system [Bruegge 91][Aral 88][Bates 83].

By providing a sequential view, we also filter out unnecessary information. Communication and synchronization are compiler generated and are not relevant to debugging and are not visible to the user. Instead of multiple program counters, there is a single program counter. We replace the parallel program with an abstraction, the source program.

The work that is the most closely related to ours is the debugging of optimized code [Hennessy 82][Copperman 90][Coutant 88][Zellweger 84] and the debugging of parallelized code by Gupta [Gupta 88] and Pineo [Pineo 91]. Debugging optimized code is similar because it requires structural mapping; however dynamic order restoration is not present in debuggers for optimized code because there is no parallelism. The differences between our work and previous work on debugging parallelized code is explained in detail in CHAPTER 4.

An alternative to the approach we pursue for debugging parallelized programs is to compile the sequential source program that the user writes for a single processor, and use conventional single processor debugging tools[Cft 90][Tseng 89]. This approach does not work when the execution environment on the sequential and parallel machines are not identical, when the program uses more memory than can be accessed by a single processor, and when the program runs too slowly on a single processor.

Another approach is a compromise between providing source-level debugging and making the user debug the parallel program. A debugger can do structural mapping, so that the user can use line numbers and variables names from the source program, but still expose the parallelism [Cft 90].

1.5 This presentation

In CHAPTER 2, we give some background information on debuggers and parallelizing compilers. We define a notation for specifying debuggers and define correctness for a

source-level debugger. CHAPTER 3 defines the scope of the problem we are solving and outlines the overall structure of a compiler and debugger based on thread splitting. CHAPTER 4 describes the thread splitting transformation and its debugger. CHAPTER 5 describes distribution for loop based parallelism and its debugger. CHAPTER 6 integrates the information in the previous two chapters to present a complete debugger for a compiler that does block and cyclic distributions of data and computation. CHAPTER 7 identifies the limitation of the thread splitting approach. Chapter 8 summarizes the thesis and identifies contributions and future work.

CHAPTER 2 Background

This chapter provides some background for our work. In Section 2.1, we describe the execution models for sequential and parallel programs. Section 2.2 describes the structure of a parallelizing compiler. Section 2.3 introduces the notation that we use to specify debuggers and defines correct behavior for source-level debugging. The notation is used throughout the rest of the thesis.

2.1 Sequential and parallel programming models

The single processor language used in examples in this thesis is the C language; the results apply for any conventional imperative language such as FORTRAN or Pascal. Programs with pointers, procedure call, and recursion are allowed. Program flow graphs must be structured. That is, there cannot be a jump into the middle of a loop. Baker[-Baker 77] describes a method for generating structured programs from a reducible flow graph and any flow graph can be made reducible by duplicating code[Aho 86].

The model of execution is a distributed memory MIMD computer. Processor names, also called processor indexes, are n dimensional vectors. The dimensionality n is chosen to match the topology of the interconnection network. Parallel programs are expressed as a collection of sequential programs, one program per processor. One process is created on each processor at the start of the program, no processes can be created after that. The programmer assumes that the processes execute asynchronously on separate processors; no assumptions can be made about the relative progress of execution of processes except for the explicit synchronization.

The topology and connectivity of the processor array are unimportant for debugging. Our examples use linear processor arrays and 2 dimensional tori with nearest neighbor connections. The sequential language is augmented with primitives for communication.

Communication must be via **send** and **receive** statements that have blocking semantics; that is, a send to a processor with a full buffer or receive from a processor that has not sent the data yet causes the processor to stall until the communication action can be completed. The amount of buffering for communication does not affect debugging.

In our examples, we use **send** and **receive** statements that move individual data items. A **send** statement has two arguments, a distance vector and a value to be sent. The receive statement also has two arguments, a distance vector and a place to store the value received. A distance vector is an offset which can be added to the index of the processor executing the communication action to obtain the index of the other processor in the communication action. On a 2D array with a NEWS grid (North, East, West, and South), the processor to the west of processor (i, j) is processor (i, j-1), so an offset of (0,-1) is used to send to the west and an offset of (0,1) is used to receive from the east. If we want to send to the east, then the sender uses an offset of (0,1) and the receiver uses an offset of (0,-1). The processor to the north of processor (i, j) is processor (i+1, j). If we want send data to the north, the sender uses an offset of (1,0) and the receiver uses an offset of (-1,0). If a processor array supports communication to non-neighboring processors, then a distance vector can have a magnitude greater than one.

A mod function is applied to the result of adding the processor index and distance to "wrap-around" processor indexes. For example, if a 1-dimensional array has 5 processors, and processor 0 executes a **send** with a distance of (-1), then it is sent to processor 4.

For convenience in writing programs, we also use **sendn** and **receiven** which are variants of **send** and **receive**. These constructs have the same semantics as **send** and **receive** except that they do not use the wrap-around connections of the processor array. If the processor index plus the distance would go outside the bounds of the processor array, then the communication action is not executed. Reusing our previous example, if processor 0 executes a **sendn** with a distance of (-1), the **sendn** is not executed. The **sendn** and **receiven** are useful when we want the processors on the ends of the array to behave differently from the rest of the processors.

2.2 Parallelizing compiler

The parallelizing compiler, or parallel program generator, takes a program written in some high level language and outputs a parallel program that can be executed on a parallel machine.

A typical parallelizing compiler has three phases: restructuring, parallelization, and single processor compilation. The first phase is *restructuring*, which consists of source to source transformations that change the structure of program constructs like loops and conditionals. Both the input and output are sequential programs. Loop interchange and strip mining[Padua 86] are some examples of restructuring transformations. Restructuring is used to increase the locality of reference, increase the potential for parallelism, and reduce the communication.

The next phase is *parallelization*, where the operations and data of the program are mapped to processors to parallelize execution. The input is a single program, with either

sequential semantics or possibly parallel constructs like `doall` and `cobegin`, and the output is a parallel target program that implements a mapping of work to processors. The output can be either a single program which can be used for all processors or multiple programs, where each processor gets its own program. Most compilers emit a single program.

If the input to parallelization has a `doall` or a sequential loop construct, the output is a program where each processor executes its assigned set of iterations of the loop. As an example, the sequential program in FIGURE 1-1 is the input to parallelization and the parallel program below it is the output.

There are two features of the parallelization phase that are important to the rest of this work. First, the source and target of parallelization perform the same computations; no changes to the algorithm are made. Second, the parallelization phase preserves the order of execution of operations in a single thread. In parallelization, we map operations to processors, but we do not change the computation or the order of execution of operations.

The final phase is single processor compilation. The tool chain for single processor compilation consists of a compiler, assembler, and linker. The result of single processor compilation is an executable which can be loaded into the memory of a processor.

The result of each of the phases of compilation is a version of a program with the same meaning as the source, but possibly in a different representation. The source version is a program written in a high-level language with parallel constructs. After parallelization, the result is a program that is still in a high-level language, but with the distribution explicit. After single processor compilation, the result is a machine language version. All of these versions of the programs have the same meaning, but only one is directly executable by the processor itself.

2.3 Semantics of source level debugging

In this section, we introduce a notation to describe debuggers, and use it to define the correct behavior of a debugger for sequential programs. This notation is used to define debuggers in the rest of the thesis. A debugger is a tool that can be used to execute a program, inspect and modify its state, and set breakpoints. A debugger is called *source-level* when it executes the machine language version of a program, but makes it appear that the source version of the program is executing.

Section 2.3.1 describes the basic functionality of a debugger and the notation used to define debuggers. In Section 2.3.2, we define correct behavior for a source level debugger. We don't need to introduce parallelism to define the behavior of a source-level debugger so to simplify the discussion, the definitions in Section 2.3.1 and Section 2.3.2 only apply to a debugger for a sequential program running on a sequential machine. In Section 2.3.3, we extend the notation to parallel programs and machines. Section 2.3.4 describes how to compose debuggers.

2.3.1 Debugger notation

The basic functionality of a debugger includes the ability to set breakpoints, examine variables, modify variables, and report the current location. The commands for these actions are: `set`, `examine`, `where`, and `run`. The `set` command changes the value of a variable. Its takes two arguments, the name of the variable and the value to change it to. The `examine` command inspects the values of variables. Its argument is the name of the variable and the debugger displays its value. The `run` command executes the program until a breakpoint or exception is reached or until the program terminates. The `run` command takes two arguments, the program to be executed and a list of breakpoints. A program may be single stepped by running with a breakpoint list of the special token `ALL`. The `where` command determines the current location in the source program, which is the next statement to be executed. The command doesn't take any arguments, and the debugger displays a unique label that identifies the statement. For a high-level program, the label can be thought of as a line number in the program.All of the commands implicitly take the program state as an argument.

A program state consists of the label of the statement about to be executed and the values of user variables. The part of the state that contains the values of variables can be thought of as a binding between names and values. If the variable is a scalar variable, the name is just an identifier. If the variable is an array element, then the name is a an identifier and array subscripts. Examining a variable looks up the value bound to the name, while modifying a variable replaces a binding of a name and value with a new one.

Rather than specify the entire debugger, which includes a user interface and other extraneous details, we define a kernel that processes our small set of commands. We call his the debugger function or D function. The functionality of a D function is:

D:

$$\text{Dstack} \times \text{Command} \times \text{State} \times \text{Program} \times \text{Bpts} \times \text{VarName} \times \text{VarValue} \rightarrow \text{State} \mid \text{VarValue} \mid \text{Location}$$

The Command argument is any one of the commands described previously. The rest of the arguments for the D function are arguments for the particular command. Some commands do not need all the arguments for the D function (e.g. `run` takes a program state, program, and breakpoint list but does not need a variable name or value). In that case, a dummy value of $\perp$ is used for the unnecessary arguments (e.g. $D(\perp, \text{run}, s, p, (3), \perp, \perp)$). The Dstack makes it possible to compose debuggers; we delay an explanation of this until Section 2.3.4. The type of the value returned by a D function depends on the command. Both the `set` and `run` commands create new program states. The `examine` command returns a VarValue, the value of a variable. The `where` command returns the location in a program.

In FIGURE 2-1, we define the behavior of a debugger with a *base* debugger where the source and target programs are the same (no compilation is necessary). The base debugger uses an *interpreter* to directly execute the program.

To model the behavior of the program, it is assumed that there is an interpreter function, I_L, where $I_L(p, s, b)$ returns a new state that is the result of executing a program p in language L on a state s until a breakpoint in the list b is reached or until the program terminates. In the new state, the current label and the values of any variables assigned to by the statements executed are changed. If the language L is the machine code, then I_L models the behavior of the processor and the resulting state reflects executing machine instructions. If L is a high level language, then I_L can be an interpreter and the resulting state reflects executing statements of the source language. Most of the details of the semantics of the language are orthogonal to the issues that we wish to address about debugging. For this reason, all that information is hidden inside the program state and the I function.

In the base debugger, the *label_of* function extracts the current label from a program state. The *access* function is used to lookup the value of a variable in a state. The arguments are a program state and the name of a variable. The *update* function is used to modify the value of a variable in a state. It takes a program state, a variable name, and a variable value and returns a new state where the variable is bound to the new value.

FIGURE 2-1 Definition of the base debugger

```
BaseDebugger(Dstack,Command,State,Program,Bpts,VarName,VarValue)
{
if (Command == "where") {
    return label_of(State)
if (Command == "examine") {
    return access(State, VarName)
if (Command == "set") {
    return update(State, VarName, VarValue)
if (Command == "run")
    return I_L(Program, State, Bpts)
```

2.3.2 Correctness of source-level debuggers

For correctness, we require that the behavior *observable* by the user be the same whether the base debugger of FIGURE 2-1 is used with a program p written in language L, or a source-level debugger is used with a compiled version of the program p.

When defining correctness, we can't simply require that all the input/output behavior of the base debugger function and the source-level debugger function be the same. While the meaning of the two programs are the same, they are different versions and hence the program states are different.

To decide if two debuggers provide equivalent behavior, we require that if we start with program states that are indistinguishable when using debugger functions to examine

them (using `where` and `examine` commands) then executing a command that modifies the program state (`run` and `set`) also leads to indistinguishable program states. We first introduce definitions for consistency and congruency to define correctness. *Consistency* is used to compare program states.

Definition 2-1

Two states s_1 and s_2 are *consistent* for debuggers D_1 and D_2 (consistent(s_1, s_2, D_1, D_2)) if:
$D_1(\perp, \text{where}, s_1, \perp, \perp, \perp) = D_2(\perp, \text{where}, s_2, \perp, \perp, \perp)$ and
$\forall n\ (D_1(\perp, \text{examine}, s_1, \perp, n, \perp) = D_2(\perp, \text{examine}, s_2, \perp, n, \perp))$.
Usually the debuggers are obvious from the context. In this case we omit them and just say states s_1 and s_2 are consistent or consistent(s_1, s_2).

If two states are consistent for a pair of debuggers, then we cannot distinguish the two states by using the debuggers to examine them. We use the $\forall$ because the condition on examining variables must be true for all possible variable names.

Now we can use the consistency of states to define *congruency* of debuggers.

Definition 2-2

Two debuggers D_1 and D_2 are *congruent* for programs p_1 and p_2 (congruent(D_1, D_2, p_1, p_2)) if:

$\forall (s_1, s_2)$
consistent$(s_1, s_2) \rightarrow$
$\forall b$consistent$(D_1(\perp, \text{run}, s_1, p_1, b, \perp, \perp), D_2(\perp, \text{run}, s_2, p_2, b, \perp, \perp))$
and
$\forall (n, v)$ consistent$(D_1(\perp, \text{set}, s_1, p_1, \perp, n, v), D_2(\perp, \text{set}, s_2, p_2, \perp, n, v))$

We define two debuggers to be congruent for two particular programs if execution of identical commands on consistent states yields a new set of consistent states. The two commands for modifying states are `run` and `set`. For the `run` command, we use $\forall$ because the condition must be true for every possible set of breakpoints. For the `set` command, we use$\forall$ because the condition must be true for every possible combination of variable names and values.

Now that we have a way of comparing the behavior of debuggers, we can define correctness for a source-level debugger.

Definition 2-3

A debugger function D is a *correct* source-level debugger for a transformation Δ when:
$\forall p$

congruent$(D, \text{basedebugger}, \Delta(p), p)$

where basedebugger is the debugger defined in FIGURE 2-1. A transformation is a function that inputs one program and outputs another program. The two programs have

the same meaning, but are in different representations (high-level language and machine code). In this definition, p is the source program and $\Delta(p)$ is the target program.

Intuitively, a source-level debugger is correct if its behavior while executing the target program is the same as the behavior of the base debugger while executing the source program. In the rest of the thesis, when we use the words correct and incorrect to describe the behavior of a debugger, we are comparing it to the behavior of the base debugger. Correct behavior has also been called *expected* behavior by Zellweger[Zellweger 84].

Some transformations change the behavior of the program so much that there cannot be a D function. This problem exists for debuggers for sequential machines as well as for parallel ones. For example, it might not be practical to allow the user to modify variables that are part of a common sub-expression[Hennessy 82]. Examining a variable may not be possible if a dead store (a dead store is a store to a variable that is not used later in the program) that modifies the variable has been eliminated, and single stepping may not be possible if an optimization removes an "unnecessary" loop. An example from parallelization is when a compiler transforms a sequential algorithm that computes the sum of an array into a parallel one which does the operations in a different order and doesn't compute the same intermediate values.

Just because one aspect of the program is not observable by the debugger does not mean that a debugger cannot be used at all. If a variable cannot be examined because a dead store was eliminated, then the debugger should not allow the user to inspect the variable; inspecting other variables should not be affected. Furthermore, the debugger should never give misleading information. The debugger should either behave the same as a correct source-level debugger or it should indicate to the user that correct behavior is not possible. Zellweger calls this *truthful* behavior. We call transformations that have a correct source-level debugger *fully debuggable* and transformations that do not *partially debuggable*.

2.3.3 Extending the debugger definition to parallel programs

Only a small number of changes are needed to extend the notation from the previous sections to cover parallel machines as well. The state for a parallel program is a set of states, one for each processor. Each state contains the current point in the program for the processor and the value of that processor's variables. A program is a set of single processor programs. A D function for a parallel program is a set of D functions for single processors. When a debugger for a parallel program must perform an action on a processor, it selects the debugger and state for that processor from the sets. For example, if the debugger wants to examine the value of a variable in processor 3, it extracts the state of the third processor from the set of states, extracts the debugger for processor 3 from the set of debuggers, and applies the debugger to that state to lookup the value of the variable requested. If the debugging action modifies a state, as is done by the set command, the resulting state is merged back in to the set of states that is considered the current state.

2.3.4 Composing *D* functions

We construct a compiler by composing code transformations; we also need a way to compose the *D* functions for those transformations to construct a debugger. If the compiler translates the code from source to target in several steps, we want to build a debugger for each step individually, then construct the debugger for the entire transformation by putting together the debuggers for those steps. For example, some C++ compilers first translate the C++ code to C code, then translate the C code to assembly code. We could build a debugger for such a system by building a debugger for C code, building another debugger that assumes the source level is C++ and the target level is C, and then put together those two debuggers to construct a new one where the source level is C++ and the target level is machine code.

A stack of debuggers comprises individual debuggers that translate requests from their source level to their target level. Requests are passed down the stack to the machine level and results are passed back to the top. One debugger is at a lower level than another if it is closer to the machine level in the debugger stack. The lowest level debugger is called the base debugger, which directly executes commands.

Each debugger, except the base debugger, assumes that it can use a lower level debugger to manipulate the target state. In effect, a debugger just translates commands to manipulate source-level objects into commands to manipulate target-level objects.

For example, if a program transformation replaces all occurrences of the variable `i1` with `i2`, then the debugger for that transformation translates requests to examine the variable `i1` to a request to examine the variable `i2` and passes on all other requests unchanged. The same applies for the modifying variables. The *D* function for the renaming transformation is called `d_i1_i2` and can be found below. The function `first` selects the first *D* function on the debugger stack. The function `rest` returns the debugger stack with the first *D* function removed. The function `apply` applies a function to a set of arguments. It has the same semantics as the Common Lisp[Steele 84] function of the same name. If the variable `foo` contains the function `bar` then `apply(foo,1,2,3)` is the same as `bar(1,2,3)`.

The first argument of `d_i1_i2`, `dstack`, is the debugger stack. The debugger stack is a list of *D* functions. The top of the stack is the *D* function that the current *D* function can use to manipulate its target state. Each *D* function peels off the top of the stack when calling the lower level *D* function. The new top of stack for the lower level debugger is the *D* function below it in the stack. When a debugger calls a lower level debugger, it applies the lower level debugger to a set of arguments. The first argument passed the *D* function is the new debugger stack with the first element removed.

The lowest level debugger on the stack (the base debugger) must perform the commands on the real target state; it does not pass on the command to a another debugger.

```
d_i1_i2(dstack,command,state,program,bpts,name,value)
{
    if (command == run) {
        return apply(first(dstatck),rest(dstack),
                        run,state,program,bpts,⊥,⊥)
    }
    if (command == where) {
        return apply(first(dstack),rest(dstack),
                        where,state,⊥,⊥,⊥)
    }
    if (command == examine) {
        if (name == "i1")
            return apply(first(dstack),rest(dstack),
                            examine,state,⊥,"i2",⊥)
        else return apply(first(dstack),rest(dstack),
                                examine,state,⊥,name,⊥);
    }
    if (command == set) {
        if (name == "i1")
            return apply(first(dstack),rest(dstack),
                            set,state,⊥,"i2",value)
        else return apply(first(dstack),rest(dstack),
                                set,state,⊥,name,value);
    }
}
```

As an illustration, assume we compose the debugger defined above, `d_i1_i2`, and a similar debugger for a transformation that renames the variables `i2` to `i3`. The base debugger is called `b`. The debugger stack for this set of transformations is (`d_i1_i2`, `d_i3_i3`, `b`). If the user wants to examine the variable `i1`, then the debugger would be applied from the top of the stack as follows:

```
d_i1_i2((d_i2_i3,b),examine,state,⊥,⊥,"i1",⊥))
```

The debugger `d_i1_i2` would pass the request to the next debugger on the stack as follows:

```
d_i2_i3((b),examine,state,⊥,⊥,"i2",⊥)
```

The debugger `d_i2_i3` would pass the request to the base debugger:

```
b((),examine,state,⊥,⊥,,"i3",⊥)
```

The base debugger would look up the value of `i3` and pass it back up to `d_i2_i3`, which would return it to `d_i1_i2`, which would return the value which is displayed to the user.

2.4 Summary

The models of execution for parallel and sequential machines were introduced. The structure of a parallelizing compiler has been described. A notation for specifying debuggers has been introduced and it was used to define the correct behavior for a source-level debugger.

CHAPTER 3 Approach

In this chapter, we describe our approach for source level debugging. We focus on the parallelization phase of compilation, since it spans the gap between sequential and parallel programs. Parallelization requires both *structural mapping* and *dynamic order restoration*. Dynamic order restoration is needed because parallelization converts the program's single thread of control to multiple threads of control, hence out of order execution is possible. Structural mapping is needed because the mechanics of parallelization require that the program structure be altered.

We attack these problems separately by dividing the parallelization phase of compilation into two parts. In the first phase, we restructure the code to expose all the structural changes necessary for parallelism. In the second phase we parallelize the code by extracting multiple programs from a single processor program.

In Section 3.1, we define the scope of the problem we solve. In Section 3.2, we review the basic functionality of what a debugger for parallelized programs should do. This is followed in Section 3.3 by an introduction to our methodology for constructing debuggers.

3.1 Problem scope

If compilation parallelizes the program but does not change the computation or the order of execution of operations on a processor, then the same values are computed in source program order and it is possible to build a debugger. If compilation alters the computation (e.g. converting a sequential reduction into a parallel reduction) or changes the order of execution of operations, then it may not be possible to construct a debugger, depending on the degree to which the computation has been changed. In this thesis, we only study the problem of debugging when the computation has not been changed.

As we described in the previous chapter, compilation can be divided into three phases: restructuring, parallelization, and single processor compilation. A debugger for the compiler can be thought of as the composition of debuggers for the individual phases. Restructuring transformations and single processor compilation are sequential to sequential transformations and can change the computation and the order of operations. Parallelization converts a program from sequential to parallel but does not alter the computation.

Each of these compilation phases can present interesting challenges to the debugger writer. Sequential to sequential transformations, whether they are applied by a restructurer or an optimizing compiler for a sequential language, have received a great deal of attention[Feiler 82][Warren 78][Zellweger 84][Coutant 88][Brooks 92][Copperman 90][Hennessy 82].

In this thesis, we only study the debugging problems associated with the transition from sequential to parallel, which occurs during parallelization[Pineo 91][Cohn 91]. Parallelization does not change the computations of a source program or the order in which they are computed. To factor out the problems associated with debugging sequential to sequential transformations, we assume that there is either a debugger for the restructuring phase, or that no restructuring is done by the compiler. Furthermore, we assume that there is a debugger for the single processor compiler.

3.2 What a debugger for parallelization must do

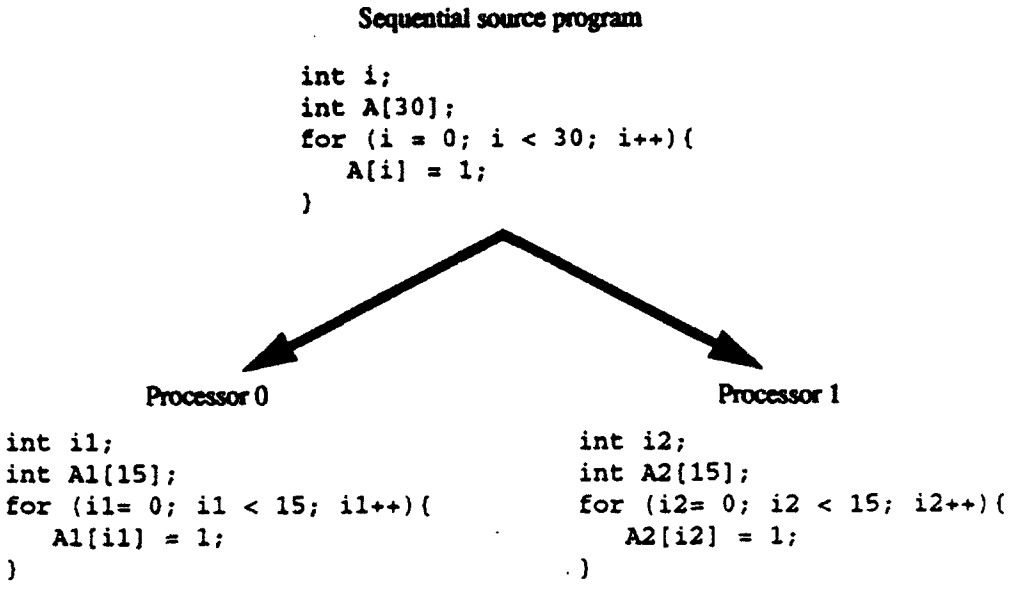
The work of the *D* function can be divided into dynamic order restoration and structural mapping. In Section 3.2.1, we describe when and why dynamic order restoration is necessary and what it must do. In Section 3.2.2 we answer the same questions for structural mapping.

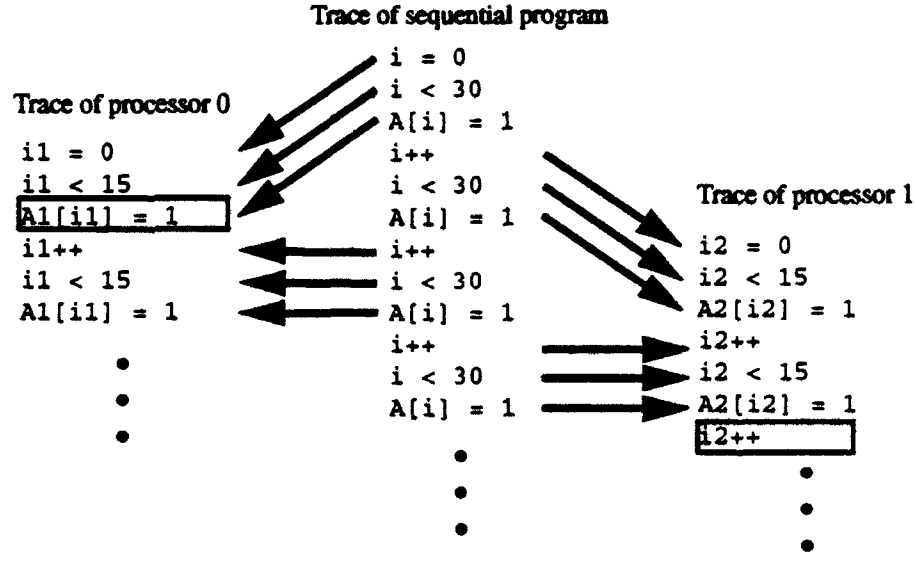
3.2.1 Dynamic order restoration

Dynamic order restoration makes it appear that the order of execution of operations is the same as the order in the source program. It is necessary whenever the compiler removes an ordering constraint that was present in the sequential program. Compilers remove ordering constraints to exploit the parallelism. For example, the semantics of most imperative languages imply that iterations of a loop are executed sequentially. If there are no dependences between loop iterations, then the compiler can generate parallel code that does not obey that ordering constraint and allows iterations to be executed in parallel.

A problem for debugging arises when we allow operations to be executed in parallel. If the program execution is interrupted because the program hit a breakpoint or the user aborted the program, it may appear that operations are executed out of source order. Out of order execution makes it impossible for the debugger to point to a position in the source program where every operation before that point has been completed and no operations after that point have been initiated. FIGURE 3-1 illustrates this point. The top of the figure is the same sequential and parallel programs as in the example used in the previous chapter. The bottom of the figure contains traces of the execution of the sequential program in the center column and traces of processor 0 and processor 1 to the

FIGURE 3-1 Parallel execution gives the appearance of out of order execution

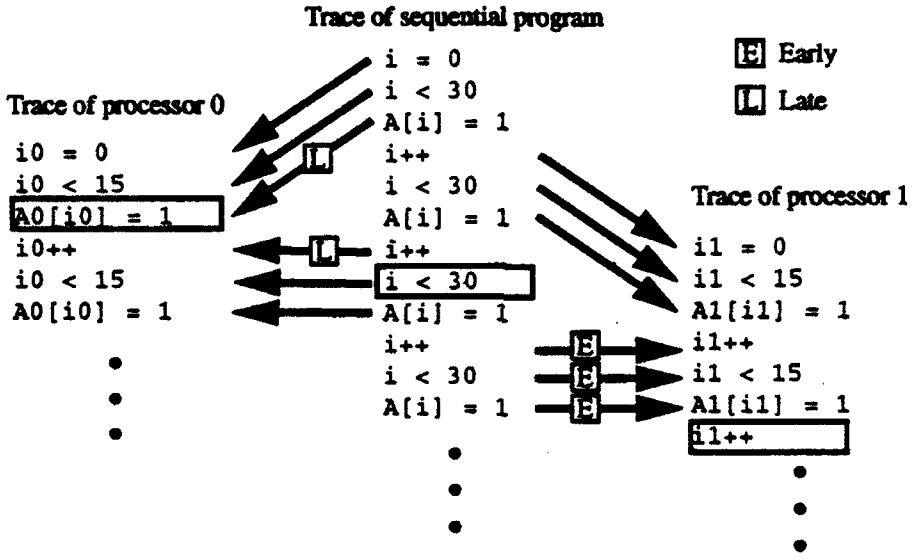




left and right. The arrows relate operations in the sequential and parallel programs. The boxes around statements in the processor 0 and processor 1 traces indicate the operations that each processor will execute next. For this particular state of the parallel program, there is no place in the sequential program trace that we can point to and say that every operation before that point has been executed and no operation after that point has been executed.

Given a point considered the current point in the source execution, operations which occur after that point in the source program, and have already been executed are called *early* and operations which appear before the current point in the sequential execution and have not been executed yet are called *late*.

FIGURE 3-2 Early and late operations



In FIGURE 3-2 we have another illustration of the same state of the program. The box around the statement in the sequential program trace indicates the statement that the debugger has chosen as the current statement (calling it the current statement implies that it has not been executed yet). The debugger could have selected other statements as the current one; the constraints on choices are explained in CHAPTER 4. Anything above the selected line in the sequential trace which has not been executed is a late operation. The arrows for those operations are marked with L's. Anything below the selected line in the sequential trace which has been executed is an early operation. The arrows for those operations are marked with E's.

If an early or late operation changes the value of a variable, and there is no dynamic order restoration, then the behavior of the debugger will be incorrect. For an early operation, the user should see the value before it was updated. However, only the value after

the update is available. In the figure, processor 1 does an early update to the array `A1`. If the user inspected `A1[1]`, they would see the new value, 1, when they should have seen the old value, 0.

If there is a late operation that modifies a variable, the user should see the value after the late operation updates the variable, which has not happened yet. In the figure, processor 0 has a late operation that updates the array `A0`. When inspecting `A1`, the user sees the pre-update value of 0 when the post-update value of 1 should be seen.

Dynamic order restoration also ensures that the user sees events such as breakpoints in the order that they would have occurred in the source program. If a late operation will cause a breakpoint when it executes, then the user should see the breakpoint associated with the late operation *before* seeing the event that made the program stop in its current state because the late operation has an earlier virtual time.

Dynamic order restoration can either force the parallel program to execute operations in source order or it can allow the program to execute operations out of order, but not let the user examine any state that has been modified by out of order operations. Choosing between these two methods depends on the situation; methods are discussed in CHAPTER 4. If the debugger doesn't force execution in source order, each command must have a set of restrictions to ensure that the user cannot see the effect of out of order execution. For the `where` command, the debugger must pick a point in the program to call the current point. This determines the set of late and early operations. The debugger *must be sure* that there are no late operations that will cause breakpoints or exceptions. For the `examine` command, the debugger must ensure that the user does not inspect a variable that is written by a late or early operation. For the `set` command the debugger must ensure that the user does not modify the value of a variable that is read or written by a late or early operation. When setting a breakpoint the debugger should ensure that a breakpoint is not set for an early operation.

Late and early operations are closely connected to roll-back and roll-forward variables as defined by Hennessy[Hennessy 82]. Variables written by early operations are roll-back variables. Variables written by late operations are roll-forward variables. Computing the sets of roll-forward and roll-back variables is sufficient if the debugger wants to determine if it is safe to examine a variable, but more information is needed if we want to determine if it is safe to modify variables or set breakpoints. Computing the late and early operations provides this extra information.

3.2.2 Structural mapping

Structural mapping relates variables and statements in the source code to variables and statements in the target code. If the program is translated, then structural mapping is needed.

Parallelization usually entails some change in the structure of the program. If the compiler implements a shared memory model on a distributed memory machine, then the names for variables used in the source program are not the same as the names used in the parallel program. In our example in FIGURE 3-1, the variable `i` has been renamed. If the elements of an array are distributed across a set of processors, as is done for the array `A` in the example, then the subscripts used to reference the array are changed.

Some variables like loop counters are replicated on all processors, even on shared memory machines. Furthermore, some variables might not have the same value as they would have had in the sequential program. In the example, the loops for both processors go from 0 to 14 while the loop in the source program goes from 0 to 29.

The structure of the code is changed as well. Instead of a single program, there is a set of programs, one for each processor. If the compiler converts a sequential loop into a parallel loop, then the loop structure must be changed so that every processor executes a subset of the iterations.

Structural mapping is a translation of names between objects in the source program and target programs. When the programmer uses the debugger to examine or modify a variable, structural mapping converts the name that is used in the source program to the name in the target program. If that variable happens to be an element of a distributed array, then structural mapping computes which processor the element resides on and the address at which it is stored. In the example of FIGURE 3-1, structural mapping would determine that array element `A[i]` resides on processor 0 if i is even and processor 1 is i is odd. It would also change the index `[i]` to `[i/2]`. If a variable is replicated, then the debugger must know which copy is the appropriate one to examine and which set of copies is the appropriate one to modify. If a variable does not have the same value as in the source program, then the debugger must know how to compute the source value from the target value to examine the variable and it must also know how to compute the target value from the source value to modify the variable. To examine the loop counter in the example, we must first decide which processor has the appropriate copy. If it is processor 0, we multiply the loop counter by 2. If it is processor 1, we multiply the loop counter by 2 and add 1. For the `where` command, we must know how to map lines in the parallel program to lines in the sequential program. To set breakpoints with the run command, we must know the inverse mapping, from lines in the sequential program to lines in the parallel program.

3.3 Debugger methodology

The debugger methodology presented in this thesis allows us to separate dynamic order restoration from structural mapping when building a debugger. As is explained later, dividing the two allows us to study the dynamic order restoration problem independent of the parallelizing transformation used by a compiler.

A debugger that does dynamic order restoration must know the total ordering of operations in the parallel program. Structural mapping requires information about the relationship of lines and variables in the source and target programs. We can separate structural mapping from dynamic order restoration in a debugger by dividing the parallelization phase of compilation into two steps: *distribution* followed by *thread splitting*. The input to distribution is the same as the input to parallelization and is called the α program. The output of distribution is another sequential program called the β program. Thread splitting inputs the β program and outputs the parallel ω program, where ω_i is the sub-program that executes on processor i. The ω program is the output of parallelization.

The operations in the β program have a 1 to 1 relationship with the operations of the ω program. Since the β program is sequential, it defines a total ordering of execution of operations of the ω program. The debugger uses the total ordering for dynamic order restoration. All of the structural effects of parallelization have been factored out and exposed in a program without parallelism (the β program). Structural effects include replicated statements and variables, altered loop structures, and distributed variables.

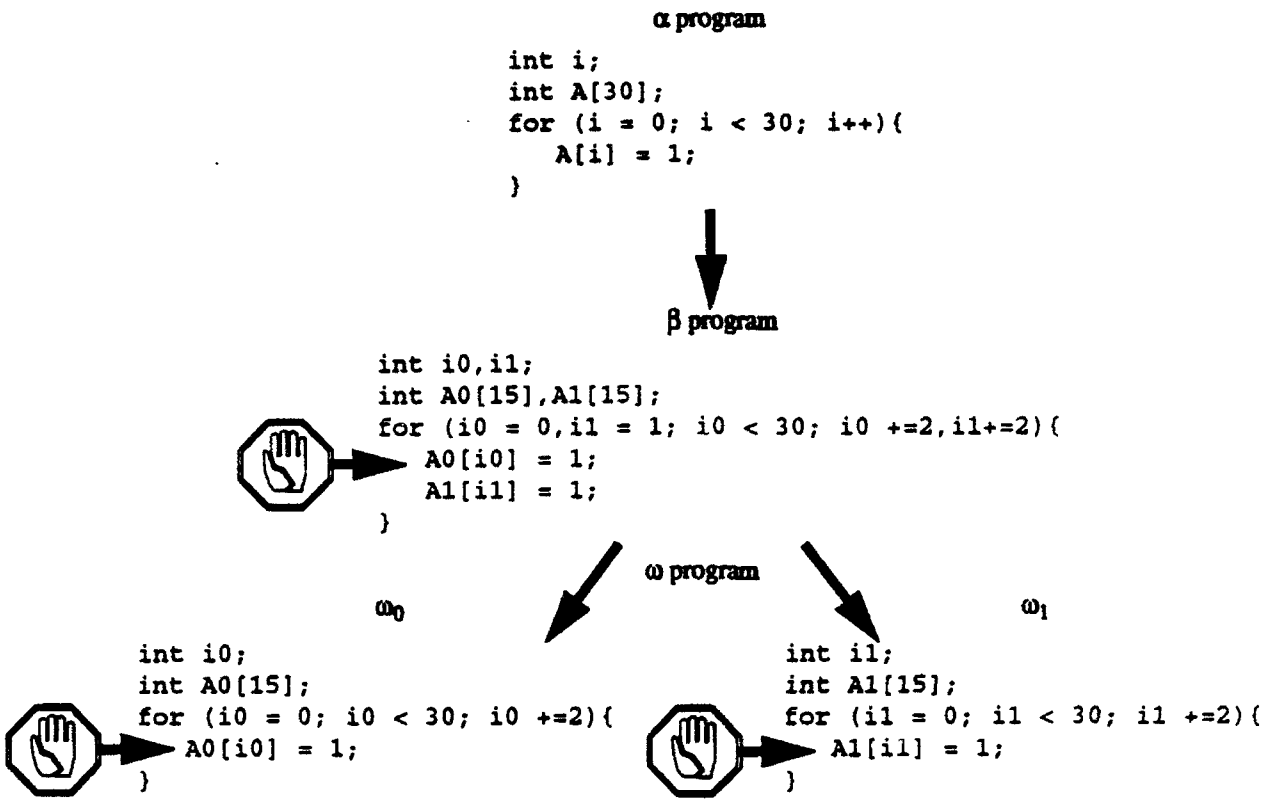
The debugger is structured in a manner similar to the compiler. There is a debugger for distribution and thread splitting; the debugger for parallelization is the composition of the two. The debugger for distribution does structural mapping but does not need to do dynamic order restoration because both the input and output are sequential. The debugger for thread splitting does dynamic order restoration and some simple structural mapping.

A debugger where structural mapping and dynamic order restoration are separated has two advantages when compared to a debugger where they are done together. The chief advantage is that it lets us isolate the functionality that manages program parallelism in a part of the debugger that is reusable for different compilers. When we construct new parallelizing transformations, a new distribution transformation is needed, but the thread splitting transformation always stays the same. Thus, the part of the debugger that handles thread splitting can be reused when the debugger is retargeted for a compiler that uses a different parallelizing transformation.

The second advantage is that structural mapping is simplified when dynamic order restoration has already been done. This is because all problems associated with out of order execution have been resolved. To illustrate this point we use the previous example of parallelization where processor 0 executes all the even iterations of a loop and processor 1 executes all the odd iterations. In FIGURE 3-3, the top program is the α program, the middle program is the β program, and the bottom program is the ω program. We explain β programs in detail in Section 4.1.1, for now it is only necessary to observe the duplication of the loop counters. Assume that the parallel program is interrupted, and both processor 0 and processor 1 are about to execute the assignment statement in the body of the loop. If the user were to ask to inspect the loop counter, it is unclear which counter is the appropriate one to show to the user by inspecting the state of the ω program. However, if we do dynamic order restoration first, we can relate the current state of execution to a point in the execution of a sequential program, which is our β program. In this case, it would indicate that the next statement to be executed is the assignment to `A1` in the body of the loop in the β program, which makes it is clear that the correct copy of the loop counter to inspect is `i1`. Doing dynamic order restoration first, in conjunction with the 1 to 1 mapping between statements in the β and ω programs, has simplified structural mapping.

The main disadvantage of separating structural mapping and dynamic order restoration is that we must construct the intermediate program called the β program, which would not otherwise be necessary. However, we do not believe that this disadvantage negates the advantages—we show in CHAPTER 5 that constructing a β program for the common distributions is straightforward.

FIGURE 3-3 α, β, and ω program



The next two chapters complete the description of debugger methodology. Chapter 4 describes the thread splitting transformation and its debugger and Chapter 5 describes distribution and its debugger.

3.4 Summary

We identified the scope of the problem we are trying to solve, which is the parallelization phase of compilation. Dynamic order restoration and structural mapping are needed for debugging parallelization. We described a methodology for factoring out dynamic order restoration from parallelization, so that it can be handled separately from structural mapping.

CHAPTER 4

The thread splitting transformation

This chapter describes the thread splitting transformation and its debugger. The thread splitting transformation converts a sequential program into a parallel program. Its source is the β program and the target is the ω program. All of the changes which are necessary for distribution are exposed in the β program. These changes include the replication and distribution of variables and the restructuring of loops. In addition, every statement in the β program is labeled with the processor on which the statement should be executed. Thread splitting uses the labeling to generate a parallel program.

The β program has an execution model of a single thread of control executing in multiple, disjoint address spaces. In generating the ω program, we extract one thread of control for each address space. There is a one to one mapping between statements in the β program and statements in the ω program. The order of execution of statements in the β program defines a total ordering of operations in the ω program, which is used for dynamic order restoration.

We first define a debugger which forces the execution order of statements in the parallel program to be in the same order as in the β program. This is a brute force method of doing order restoration. We then define a debugger that allows the parallel program to execute unrestricted. Depending on the state that the program happens to stop in, the debugger may not be able to allow the user to examine or modify some variables. The debugger must ensure that it provides truthful behavior; it cannot supply any information to the user that contradicts the behavior of a correct debugger.

We describe the thread splitting transformation and the language of β programs in Section 4.1. In Section 4.2, we define the debugger that executes statements in the same order as the β program. We describe how to allow parallel execution in Section 4.3. In Section 4.4, we discuss efficiency issues related to dynamic order restoration. Finally, we describe some work related to the issues discussed in this chapter in Section 4.5.

4.1 The thread splitting transformation

Thread splitting is a simple transformation where we extract multiple programs from a single program. Each statement of the source program is marked with a processor index label that indicates which processor should execute that statement. The semantics are chosen so that after thread splitting, we have a parallel program with the same meaning as the source program.

We first define the language of β programs and its semantics in Section 4.1.1. We then describe how code generation works for thread splitting in Section 4.1.2 and conclude with an example in Section 4.1.3.

4.1.1 The language for β programs and its semantics

The model of execution for a β program is that of a single thread of control executing in multiple address spaces, SIMD style. Each statement is annotated with the address spaces that the current statement should operate on.

The language for β programs is the language of source programs augmented with additional constructs for communication and the specification of the mapping of statements to processors. Its semantics are the same as the source program with some exceptions noted below. In the thesis, the source language is sequential so the semantics of the language of β programs are sequential as well.

4.1.1.1 Processor index labels

All statements have at least one processor index label (PIL). The value of the PIL's indicates which processors execute the statement in the parallel program. A PIL is an n element vector for an n dimensional processor array and must be a compile-time constant. If the value of a PIL is not a valid processor name (out of range, non-integer), then the PIL is invalid. Invalid PIL's can only occur if the compiler is incorrect.

A single PIL is written as a comma separated list of integers enclosed in "<>". The length of a PIL is the same as the dimensionality of the processor array. If there is more than one PIL for a statement, they are separated by commas. Some examples of statements with PIL's can be found below:

```
<1>,<2>,<3> b = 1;
<2>,<4>     c = 2;
```

There is one distinct address space for every valid value of a PIL. Variable declaration statements create one copy of a variable for every PIL for that statement. The PIL's determine which address space those variables reside in. Other statements such as assignments, conditionals, etc., always reference the copy of the variable that is in their namespace, which is determined by the PIL for that statement. If the statement has more than one PIL, side effects of the execution of the statement affect every address space named by a PIL. Assignment statements have straightforward semantics for multiple PIL's. For a statement like the following, the effect is to set both the copy of c in <2> and the copy of c in <4> to the value 2.

```
<2>,<4>  c = 2;
```

For statements with control flow, such as loops and conditionals, any side effects in the loop header or condition affect every address space named by a PIL. For example, in the statement below the side effects of the statement are to initialize the variable `i` to 0 on loop entry and to increment the loop counter for every iteration of the loop. The semantics of the statement are to initialize the copies of `i` in address spaces 0 and 1 to 0 on loop entry and to increment both copies of the loop counter for every iteration.

```
<0>,<1> for (i = 0; i < n; i++)
```

Note that having the same variable in multiple address spaces does not mean that all the copies always have the same values. It is legal to assign one copy a value, but not others. For example, the following statements can be part of a legal program.

```
<1>,<2>  int c;
<1>      c = 2;
<2>      c = 4;
```

The compiler must follow 3 rules in assigning PIL's. If they are not obeyed, then the behavior of the program is undefined. The rules are:

1. A variable cannot be referenced in address space *p* unless it has been declared in address space *p*.
2. Statements with data dependent control flow may only have multiple PIL's if the control flow is the same for all copies of the statement. For example, the result of the test of the conditional must be the same using the copy of `c` in <1>, <2>, and <3> in the statement below.

```
<1>,<2>,<3>     if (c == 2)
```

3. When one statement is lexically nested inside another, as is the case for the `if` and `for` statement, the set of PIL's of every statement inside must be a subset of the PIL's for the enclosing statement. For the program below the condition part of the `if`, must have <1> and <2> as PIL's because statements contained inside have those PIL's.

```
<1>,<2>,<3> if (a == 1)
<1>             c = 1;
            else
<2>             c = 2;
```

4.1.1.2 Rep and dist constructs

It is impossible to write programs where the size of the processor array is not known at compile-time because PIL's must be compile time constants. The implications of constant PIL's are discussed in CHAPTER 7. Two run-time variables and two constructs must be added to the source language to make it possible to create programs that are parameterized by the size of the processor array.

The variables are `_np` and `_id`. They are prefaced by an underscore to separate them from user variables. The variable `_np` contains the size of the processor array and is initialized by the run-time system. The variable `_id` contains the index of the current processor and is also initialized by the run-time system. Both the size of the array and processor indexes can be *n* dimensional numbers for *n* dimensional processor arrays. In this case `_np` and `_id` are *n* element arrays.

The constructs are `rep` and `dist`. They are purely a shorthand for writing programs that are compact and independent of the size of the processor array. The `rep` is used to create a statement with multiple PIL's and the `dist` is used to create multiple copies of a statement, each with a different PIL. Loop-like control is used to generate offsets, which are added to the PIL's of statements in the body to generate distinct PIL's for every copy. The syntax of the two statements is:

`rep` *var* `=` *low* `to` *high* `with` *offset body*

`dist` *var* `=` *low* `to` *high* `with` *offset body*

Tokens in italics are syntactic variables. The *var* is a variable name. The *low* and *high* are expressions that are functions of constants, variables defined by a `dist` or `rep`, and `_np`. The *offset* is a PIL that is a function of constants, *var*, and `_np`. For a `dist`, *body* is a statement or block of statements. For a `rep`, *body* is a single statement. The scope of *var* is *body*. The *var* may be referenced in *low* and *high* expressions in `rep` and `dist` statements in the body, but not in any other statement of the body.

The `rep` and `dist` are special statements which are not executable and do not have their own PIL's. They do change the semantics of executable statements contained in the body. A `dist` is used to create multiple copies of a block of statements which can then be distributed across a set of processors. One copy of the body is created for each possible value of the `dist` variable, which is bounded from below and above by *low* and *high* with a step size of 1. The copies of the block are executed sequentially in the order that they are generated. The *offset* is added to the PIL of every statement in the body. The PIL's in the body can be any value. The *offset* can have a different value for every copy of the body because it is a function of *var*. As an example, in the `dist` below, three copies of the body are created. The value of the offset vectors are <0,0>, <1,0>, and <2,0>.

```
dist b = 0 to 2 with <b> {
<0> C = 1;
<0> D = 2;
}
```

When we expand the above `dist` we obtain:

```
        {
<0>        C = 1;
<0>        D = 2;
        }
        {
<1>        C = 1;
<1>        D = 2;
        }
        {
<2>        C = 1;
<2>        D = 2;
        }
```

A **rep** is used to generate multiple PIL's for a single statement. One PIL is generated for every possible value of the **rep** variable by adding the current value of the offset to the PIL for the statement that is the body of the **rep**. In the statement below, the **rep** creates 3 PIL's for the statement.

```
        rep a = 0 to 2 with <a>
<5>     b = 1;
```

If we expand the above **rep**, we obtain:

```
<5>,<6>,<7> b = 1;
```

A **rep** only generates the PIL's for a single statement, even if that statement contains other statements. For example, if the body of a **rep** is a **for** statement, then multiple PIL's are generated for the **for** statement, but the body of the loop is not changed. This makes it possible to replicate loop control across a set of processors but still distribute the body of the loop. If we want to replicate the body of the **for** as well, then a **rep** must be used for every statement of the body. In the example below we want to replicate the **for** statement and its body on every processor of a 2 processor linear array. Each statement must have a **rep**.

```
        rep a = 0 to 1 with <a>
<0>     for (i = 0; i < n; i++) {
           rep a = 0 to 1 with <a>
<0>        b++;
           rep a = 0 to 1 with <a>
<0>        c++;
        }
```

If we were to expand the **rep**'s in the above program we would obtain:

```
<0>,<1>  for (i = 0; i < n; i++) {
<0>,<1>     b++;
<0>,<1>     c++;
         }
```

To simplify code generation, we require that the program for each processor be identical. This occurs under the following condition. If we expand all the `rep` and `dist` constructs of a program, then for every statement *s* in the program and for every processor index *p* in the processor array, there should be exactly one copy of statement *s* in the expanded program that has *p* as one of its PIL's. We list two examples of valid programs below. In the first program, there is exactly one copy of every statement for every processor;

```
dist a = 0 to _np[0]-1 with <a,0>
   dist b = 0 to _np[1]-1 with <0,b>
<0,0>    i = 1;
```

Expanding this program for a 2x2 processor array yields:

```
<0,0>    i = 1;
<0,1>    i = 1;
<1,0>    i = 1;
<1,1>    i = 1;
```

In the following program, there is less than one copy of each statement for each processor, but each copy has multiple PIL's.

```
dist a = 0 to _np[0]-1 with <a,0>
   rep b = 0 to _np[1]-1 with <0,b>
<0,0>    i = 1;
```

Expand the `rep` and `dist` constructs for a 2x2 processor array yields:

```
<0,0>,<0,1> i = 1;
<1,0>,<1,1> i = 1;
```

The effect of this restriction is that every statement of the β program is executed in every address space. Of course, in some situations we want only a single processor to execute a statement. The restriction does not prevent it. In FIGURE 4-1 there is an example of a β program that places a different statement on every processor, and below it is the program rewritten so that the same code can be used on every processor.

This restriction simplifies code generation because it allows us to emit a single program for all processors; it does not place an undue burden on the compiler writer. All the compilers that we are familiar with already generate a single program for all processors[Ribas 90][Cft 90][Sussman 91][Chatterjee 91][Tseng 89]. We believe that most compilers do not generate different code for each processor because it makes it difficult to make a program that can be run on a varying number of processors. Furthermore, it is much more expensive to compile and load many programs versus compiling and loading a single program for all processors.

4.1.1.3 Communication

During the course of a computation, data must be moved from one processor to another. This data movement must be expressed in the β program as a movement between

FIGURE 4-1 β program that maps different statements to each processor and an equivalent β program that maps the same statements to each processor

```
<0>   A = 1;
<1>   A = 2;
```

```
      dist b = 0 to _np[0]-1 with <b>
<0>   if (_id[0] == 0)
<0>      A = 1;
      dist b = 0 to _np[0]-1 with <b>
<0>   if (_id[1] == 1)
<0>      A = 2;
```

address spaces. For this reason, we include **send** and **receive** primitives in the language of β programs.

In the β program, communication must be inserted into the program so that a **send** is always executed before its corresponding **receive**. If the **send** is not executed before the **receive**, then the program may deadlock during debugging. Furthermore, the PIL's of the sending and receiving processors must match the connectivity of the processor array. In our mesh connected 2D array, the value sent in the statement below:

```
<3,5> send((0,-1),7)
```

can only be received by a statement with a PIL of <3,4> that does a receive with direction (0,1).

To simplify our examples, we sometimes group several communication actions together into a single primitive. For example, instead of constructing a broadcast operation out of send and receives, we use a broadcast primitive.

4.1.2 Code generation

When we apply thread splitting to the β program, we extract a program for every processor. The program for processor p consists of all the statements in the β program with a PIL of p.

We require that there is exactly one copy of every statement in the β program for each processor so code generation is straightforward; the program for a processor is the same as the β program with the PIL's, **rep**, and **dist** constructs removed.

As an example, if we apply thread splitting to the second program in FIGURE 4-1, we would obtain the following ω program:

```
if (_id[0] == 0)
   A = 1;
if (_id[1] == 1)
   A = 2;
```

The β program behaves as if there is a single thread of control executing in multiple, disjoint address spaces. After thread splitting, the resulting ω program is a set of independent threads each executing in its own address space. In the second program in FIGURE 4-1, the semantics of the language for β programs imply that both copies of the first statement are executed simultaneously. In the ω program, there is no synchronization to enforce this constraint; processor 0 could execute the statement before, at the same time, or after processor 1 executes its copy of the same statement.

4.1.3 Example: loop pipelining

In this section, we present an example that demonstrates the use of PIL's, `rep`, `dist`, and communication.

We start with the α program found in part (a) of FIGURE 4-2. We use loop pipelining to distribute the work to processors. In loop pipelining, the work of a body of a single iteration is spread across several processors[Sussman 91]. Intermediate values in the loop iteration are passed from processor to processor; part (b) of FIGURE 4-2 illustrates the flow of data between processors.

Part (c) of FIGURE 4-2 shows the β program; the program to the right is the same program where the `rep` and `dist` have been expanded. Each statement has been annotated with the appropriate PIL. The variable declaration has been placed inside a `rep` so that each processor can have its own copy. The `for` loop is mapped to all processors, so it is placed inside a `rep`. Each processor must follow a different execution path through the body of the loop, so the body is placed inside a `dist`. Since the value of `A` is produced in one processor and consumed in another, its value is transferred between processors using `send` and `receive`.

This program illustrates the rules for using PIL's and communication. For communication, the sends must appear before receives. Processor 0 sends a value and processor 1 receives it, so the `dist` is written so that the copy of the code for processor 0 is before processor 1. If the program were changed so that processor 1 sent the value to processor 0, then the `dist` would have to be changed so that the copy for processor 1 appears before the copy for processor 0. This can be done by changing the offset in the `dist` to <1-b>.

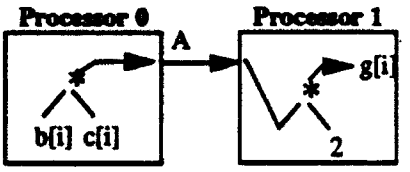
Rule 1 for PIL's requires that a variable must be declared in every address space in which it is used. This rule is satisfied by our example and any other valid program because every statement must have a copy on every processor; thus every variable is by default declared in every address space.

FIGURE 4-2 Thread splitting example for loop pipelining

(a) α program

```
int i,A,b[30],c[30],g[30];
for (i = 0; i < 30; i++) {
   A = b[i] * c[i];
   g[i] = A * 2;
}
```

(b) mapping of operations to processors



(c) β program and its expanded version

```
      rep a = 0 to 1 with <a>
<0>      int i,A,b[30],c[30];
      rep a = 0 to 1 with <a>
<0>      for (i = 0; i < 30;i++)
            dist b = 0 to 1 with <b>
<0>            if (_id[0] == 0) {
<0>               A = b[i] * c[i];
<0>               send((1),A);
               } else {
<0>               receive((-1),A);
<0>               g[i] = A * 2;
               }
```

```
<0>,<1>  int i,A,b[30],c[30];
<0>,<1>  for (i = 0; i < 30;i++) {
<0>         if (_id[0] == 0) {
<0>            A = b[i] * c[i];
<0>            send((1),A);
            } else {
<0>            receive((-1),A);
<0>            g[i] = A * 2;
            }
<1>         if (_id[0] == 0) {
<1>               A = b[i] * c[i];
<1>               send((1),A);
            } else {
<1>               receive((-1),A);
<1>               g[i] = A * 2;
            }
         }
```

(d) ω program

```
int i,A,b[30],c[30];
for (i = 0; i < 30;i++)
   if (_id[0] == 0) {
      A = b[i] * c[i];
      send((1),A);
   } else {
      receive((-1),A);
      g[i] = A * 2;
   }
```

Rule 2 for PIL's requires that a statement with data dependent control flow and multiple PIL's behave the same for every PIL. The `for` statement obeys this rule because we know that the copy of the variable `i` in <0> is always the same as the copy of the variable in <1>, hence the branching of the `for` is always the same in all address spaces.

Rule 3 for PIL's specifies that if one statement is nested inside another, the PIL's for the inner statement must be a subset of the PIL's for the outer statement. The body of the `for` loop contains statements with a PIL of <0> and <1>, so the `for` loop header must have PIL's that include <0> and <1>.

The result of thread splitting is shown in part (d) of FIGURE 4-2.

4.2 The *D* function for thread splitting

The β program, which is the source program for the thread splitting transformation, defines a total order of operations in the ω program. Thread splitting removes all the order constraints between processors that aren't the result of communication. If the ω program is interrupted during execution, either by a user interrupt, program exception, or breakpoint, then the state of the program is not necessarily consistent (see Definition 2-1) with any state of the β program because there may be early or late operations. The debugger for thread splitting must do dynamic order restoration to provide correct behavior (see Definition 2-3). One way to do dynamic order restoration is to force the execution of operations in the ω program to be in the same order as they are executed in the β program. Since the β program has sequential semantics, we must execute the ω program one operation at a time. We call this executing with a *sequential schedule*.

An alternative to executing with a sequential schedule is to allow the program to execute in parallel and detect when the program is not in a state that is consistent with a state of the β program. The debugger can then prevent the user from doing anything that would lead to incorrect behavior, such as examining a variable when it has been updated by an early operation.

Either method requires a way for the debugger to determine at run-time what order the β program would have executed the operations of the ω program. Executing with a sequential schedule follows this ordering when determining what statement on what processor executes next. If we allow parallel execution, then the ordering is used to determine if there are any late or early operations.

The ordering is determined by assigning virtual times to the execution of statements. By comparing the virtual times of statements, we can decide which one should be executed first. In the rest of this chapter, we will also use the term the virtual time of a processor. The virtual time of a processor is the virtual time of the statement that the processor will execute next.

In Section 4.2.1, we define the properties that a virtual time must satisfy and describe how to compute the virtual time of a statement. We will then show how the virtual time can be used to construct a *D* function which executes the program with a sequential schedule in Section 4.2.2. In Section 4.3 we show how to allow the program to execute in parallel while debugging.

4.2.1 Computing virtual times

Each execution of a statement is called an instance of a statement. The execution order of instances of the β program defines a total ordering. There is a 1 to 1 mapping between instances in the β program and instances in the ω program.

We want to assign virtual times to the *execution* of statements in the ω program so that we can determine what order they would have been executed in the β program. Each instance of a statement in the β program is assigned a virtual time greater than the virtual time of the previous instance. More formally:

Equation 4.1

$$s_1 \text{ occurs before } s_2 \text{ in the } \beta \text{ program} \Leftrightarrow \text{virtual\_time}(p_1) < \text{virtual\_time}(p_2)$$

s_1 and s_2 are statement instances in the β program. p_1 and p_2 are their corresponding statement instances in the ω program, respectively. If two statements on different processors have the same virtual time, then both of those statements are derived from the same statement in the β program.

In the following section we present a method for computing virtual times; this method satisfies Equation 4.1.

4.2.1.1 Computing virtual times for the β program

First, we describe a method for assigning virtual times to statement instances in the β program so that the virtual time of instances are strictly increasing during execution of the program. Next, we describe how we can assign the same virtual times to the corresponding statement instances in the ω program.

The virtual time of a processor is a tuple of $2n + 1$ numbers where n is the maximum level of nesting of loops in the program. It is represented below as a vector called *vtime*. A program without any loops has 0 levels of nesting and hence has one counter. A virtual time t_1 is greater than a virtual time t_2 if t_1 is lexicographically greater than t_2. Virtual time is advanced by executing statements of a program. Virtual time can be computed by following these rules:

1. Initially, *nesting* = 0
2. After entering a loop, *nesting*++
3. After exiting a loop, *vtime*[2**nesting*] = 0, *vtime*[2**nesting*+1] = 0, *nesting*--
4. After branching to the top of a loop, *vtime*[2**nesting*]++
5. For each statement, *vtime*[2**nesting*+1] = line number of the statement about to be executed

For a `dist` statement in a β program, each copy of the body must have a virtual time greater than the previous copy. A `dist` is treated like a loop where each copy of the body is a loop iteration. When entering a loop, *vtime*[*nest*] is the line number of the `dist` and *nesting* is incremented by 1. For every copy of the `dist`, *vtime*[2**nest*+1] is the value of the `dist` variable, which is strictly increasing. Since all copies of a statement in a `rep` correspond to the same statement, a `rep` does not need special treatment; every copy created by the `rep` has the same virtual time. For the program in FIGURE 4-3, the sequence of virtual times for an execution where `k > 1` is:

(2, 0, 0, 0, 0), (3, 0, 4, 0, 4), (3, 0, 4, 0, 5), (3, 0, 4, 1, 4), (3, 0, 4, 1, 5), (3, 1, 4, 0, 4), (3, 1, 4, 0, 5), (3, 1, 4, 1, 4), (3, 1, 4, 1, 5)

4.2.1.2 Computing the virtual time of a statement during execution of the ω program

When a processor stops executing, the debugger must compute the virtual time of the current statement. We want to precompute as much of the virtual time for a statement as possible. In the ω program of FIGURE 4-3, the left hand column is the *virtual time template* for each statement. A variable in the virtual time template indicates that the value should be taken from a `dist` variable or a loop counter. The value of the 1st, 3rd, 5th, ..., 2n+1 positions of the virtual time can be determined statically from the program counter and are part of the virtual time template. The value of counters for a `dist` can be determined from the processor index which is also static information, and the value of counters for loops can only be determined at run-time.

FIGURE 4-3 Sequential and parallel program

```
α program:
11:   k = 1;
12:   for(i = 0; i < 4; i++)
13:      A[i] = i;
β program:
1           rep a = 0 to _np[0] with <a>
2  <0>   11:k = 1;
3           dist b = 0 to _np[0] with <b>
4  <0>   12:   for (i=_id[0]*2;i<(_id[0]+1)*2; i++)
5  <0>   13:      A[i-_id[0]*2] = i;
ω program:
(2,0,0,0,0) 11:k = 1;
(3,b,4,i,4) 12:for (i=_id[0]*2;i<(_id[0]+1)*2; i++)
(3,b,4,i,5) 13:   A[i-_id[0]*2] = i;
```

The value of the counters that are associated with loops are the number of times the loops have executed. This can usually be determined from the value of the loop counters that already exist in the program. In the example, `i` can be used for the counter in the fourth position. Loop counts for virtual time always start by 0 and are incremented by 1. The counters for loops written by the user may not necessarily do this. In the thesis, we assume that all loop counters are normalized so that they always begin at 0 and have increments of 1.

The counter value for a `dist` construct is more difficult to determine. The `dist` variable is not a real variable that exists in the program. The processor index label of a statement inside a `rep` or `dist` is a function that maps the value of the `rep` and `dist` variables to a processor index. We want the inverse, a function that maps the processor index to the value of the `dist` variables. Since we require that every statement be mapped to every processor exactly once, we know that the processor index label func-

tion is a 1 to 1 and onto mapping to `rep` and `dist` variables and must have an inverse. We use a brute force approach to find the inverse. We compute all the possible combinations of values of `rep` and `dist` variables for a statement, and compute the associated processor index. If we put the results in a table indexed by the processor index, we can compute the `rep` and `dist` variables from the processor index. The table has a fixed size because there can only be one entry per processor. For the example in FIGURE 4-3, the `dist` variable b always has the value 0 on processor 0, and 1 on processor 1.

In the example of FIGURE 4-3, the virtual time of a processor executing the first statement is (2,0,0,0,0). If processor 0 is stopped at the second statement and the value of the loop counter is 0, then the virtual time is (3,0,4,0,4). If processor 1 is stopped at the third statement and the value of the loop counter is 3, then the virtual time is (3,1,4,3,5).

The discussion above assumes that the entire program is a single procedure. Our virtual time method can be extended to handle programs with procedure call and recursion in a straightforward manner. Within a procedure, virtual time is computed the same way. We shall call this ν_{proc} for procedure virtual time. As part of the normal calling sequence, every time a procedure is called the program saves the return address on the program stack. For each return address on the stack, we compute a ν_{proc}. The virtual time of the program, or ν_{prog} is computed by building a vector of ν_{proc}'s, where the first one is the one deepest on the stack (earliest in time), the second element is the second deepest on the stack, and so on. The virtual time is now a vector of vectors. Recursion does not change the way the virtual time is computed. Virtual times are can be ordered using a lexicographic comparison.

As mentioned in the previous chapter, program flow graphs must be reducible. This is because the virtual time scheme we presented assumes the program is structured, there must be an identifiable loop structure.

4.2.2 *D* function for thread splitting with a sequential schedule

If we execute the parallel program according to a sequential schedule, then no additional order restoration need be done; the debugger must only do structural mapping. We can execute with a sequential schedule by executing the statements in virtual time order.

Structural mapping for thread splitting is straightforward because of the simple mapping between statements and variables in the β and ω programs. The restriction that all programs of the ω program be identical simplifies the problem further. There is one copy of every variable in every name space and one copy of every statement for every processor. If the user sets a breakpoint on a statement, then we want to set a breakpoint on every copy of that statement in each address space. If the `where` command is used to inspect the current location, we use the location of the processor with the lowest virtual time because this is the statement that is executed next. If the user wants to inspect or modify a variable in the β program, they must specify which copy they want. A copy of a variable can be referenced with its name in the address space and the processor index. For example if the user wants to inspect the copy of the variable C on processor <1,2>, they would use the name (C,<1,2>).

The *D* function that executes according to a sequential schedule is listed below. The parallel program state is a sequence of states, one for each process. The argument *dstack* is a list of debugger stacks, one for each processor. The argument *program* is a list of programs, one for each processor. The *ad_space* function determines the index of the processor that a particular variable is a member. The function *min_virt_time* returns the index of the processor with the minimum virtual time. The *select* function picks one element out of a list, based on the index supplied. It is used to extract the appropriate debugger stack from the *dstack* parameter.

For every command but `run`, the debugger decides which processor this command acts on and passes on the command to that processor. If `examine` or `set` are used, then the processor selected is the one on which the variable resides. For the `where` and `run` commands, the processor selected is the one with the lowest virtual time.

For the `run` command, we repeatedly single step the program until we reach a breakpoint or the program terminates. After every step, we must determine which processor should be the next one to single step.

```
debugger(dstack,command,state,program,bpts,name,value)
{
    /* decide which processor this command acts on */
    if (command == where | command == step)
        index = min_virt_time(state);
    else index = ad_space(name);

    /* select the debugger stack, processor state
        and program for the processor */
    stack1 = select(dstack,index);
    state1 = select(state,index);
    program1 = select(program,index);
    deb1 = first(stack1);

    /* pass the command to the lower level debugger */
    if (command == run)
        /* single step until we hit a breakpoint
            or the program terminates */
        location=apply(deb1,rest(stack1),
            where,state1,program1,bpts,name,value)
        while (location not a member of bpts and
                    no exceptions yet) {
            state1 =
                apply(deb1,rest(stack1),
                    run,state1,program1,all,name,value);

            merge state1 into state
            /* decide which processor to act on */
            index = min_virt_time(state);

            /* select the new processor state, debugger stack
                and program */
```

```
            stack1 = select(dstack,index);
            state1 = select(state,index);
            program1 = select(program,index);
            deb1 = first(stack1);

            location
                =apply(deb1,rest(stack1),
                    where,state1,program1,bpts,name,value)
        }
        merge state1 into state
        return state
    if (command == where) {
        return apply(deb1,rest(stack1),
                                where,state,program1,bpts,name,value)
    if (command == examine)
        return apply(deb1,rest(stack1),
                    examine,state1,program1,bpts,name,value);
    if (command == set) {
        state1 = apply(deb1,rest(stack1),
                        set,state1,program1,bpts,name,value);
        merge state1 into state
        return state
}
```

4.3 Parallel execution while debugging

While executing the parallel program with a sequential schedule permits full debuggability, it has the drawback that execution is very slow. Another option for dynamic order restoration is to let the program run in parallel, and detect when the program in not in a state that is consistent with a state of the β program.

Parallel execution creates two problems. The first is that events such as breakpoints and exceptions may occur in the parallel program in a different order then they would have occurred in the sequential program. Virtual times are used to determine the correct order for presenting events to the user. The second problem is that when the parallel program stops, it might not be in a state that is consistent with any state of the β program. If we allow the user to examine or modify variables, then the behavior might not be correct (as defined by Definition 2-3). The debugger must detect when the state of the program is not consistent with a state of the β program and prevent the user from doing anything that will cause incorrect behavior. A debugger can also provide mechanisms that make it possible for the user to put the program into a state that is consistent with a state of the β program.

We begin with Section 4.3.1, which defines some terms. The basic functioning of the debugger is described in Section 4.3.2, which introduces the sub-components.

4.3.1 Terms

In this section we define some terms that are used in the discussion of parallel execution. The concepts of virtual time, and committing events are taken from Jefferson's Time Warp Operating System[Jefferson 85] [Jefferson 87]. An *event* is an occurrence caused by the execution of the program that requires some debugger action, like breakpoints or exceptions. The *virtual time of an event* is the virtual time of the statement that caused the event. A breakpoint or exception is *pending* if it has occurred but hasn't been presented to the user yet. *Committing an event* is the act of presenting a pending event to the user. In some situations, the debugger may not be able to let the user examine a variable, modify a variable, or set a breakpoint and still maintain correct behavior. In this case, the debugger tells the user that it cannot complete the action. We call this *disallowing* a command. In some cases, we may want to allow a processor or set of processors to execute until they reach a specific virtual time. We call this *roll forward*.

4.3.2 Debugger

The debugger functions as follows. The user starts the program, and it executes in parallel. Next, an event such as a breakpoint or exception occurs on one or more processors or the user aborts execution of the program. The debugger stops all processors and determines their virtual times. All events are marked as pending. The debugger then picks a virtual time in the execution of the β program to represent to the user as the current time; we call this the β time. For correct behavior, it must appear to the user that all operations with a lower virtual time have been executed and that no operations with a higher virtual time have been executed yet. Picking the β time is explained in Section 4.3.3. After selecting the β time, the debugger can present pending events to the user, and wait for the user to execute some commands. If the state of the program is not consistent with a state of the β program, then the debugger must disallow some commands. The method for deciding if commands should be disallowed can be found in Section 4.3.4. In Section 4.3.5, we conclude with a description of what the debugger can do to help the user if a command is disallowed.

4.3.3 Choosing the β time

In some cases, we have some flexibility about what time the debugger picks as the β time. This decision determines which operations are early and which operations are late, which in turn determines which variables may be inspected or modified and where breakpoints can be set. Possible choices range from the earliest virtual time of all the processors to the latest virtual time. If we pick the earliest virtual time, then we can be sure that there are no late operations. The debugger should never pick a time earlier than the earliest processor because it would only increase the number of early operations without decreasing the number of late operations. If we pick the latest virtual time, then we can be sure that there are no early operations. The debugger should never pick a time later than the latest processor because it would increase the number of late operations without decreasing the number of early operations.

Interrupts by the user are not tied to a particular statement of the program, hence there is some flexibility for choosing the β time when this occurs. If our only concern is to minimize the number of early and late operations, then the best time is the latest virtual time of all the processors. With this choice, there aren't any early operations, and we can roll

forward all the processors with a virtual time lower than the β time until there are no more late operations.

Eliminating all the late and early operations is beneficial because the current state will be consistent with a state of the β program. In such a state, the debugger will not disallow any commands. However, rolling forward to the latest virtual time may prevent the debugger from having a timely response to interrupts. For example, if a loop has 1,000 iterations, and each processor is assigned a block of 100 contiguous iterations, then one processor will immediately execute iteration 900 of the loop. If we chose the highest virtual time as the β time, then for an interrupt the debugger would always roll forward to at least iteration 900 in program, no matter when the interrupt occurs and how long it takes for the program to get to iteration 900.

For interrupts, we believe that a timely response is more important than being able to stop in a state where there are no early operations. Since it simple to roll processors forward, the β time should be the virtual time of the earliest processor. The user can late decide to roll processors forward to eliminate early and late operations, but they still have the opportunity to inspect earlier state.

Unlike interrupts, there is no choice in the selection of the β time when breakpoints or exceptions are pending. If the debugger is to present events in source order, the β time must be the time of the earliest pending event. Before committing the event, the debugger must roll forward all processors with late operations to make sure that there are no late events. If there are any late events, then they would have a virtual time less than the β time of the breakpoint or exception so they must be presented first. If new events occur during roll forward, those events become pending, and the new β time is the time of the earliest pending event.

To summarize, user interrupts are handled differently from events such as breakpoints and program exceptions. If the program is running and an event occurs, we stop all processors and determine their virtual time. The β time is chosen to be the time of the earliest pending event. We then roll forward all processors to that β time. Another event may occur during roll forward; if it does the debugger handles it the same as if the program were running and an event occurred. Once roll forward is complete, we commit the event by letting the user know that a breakpoint or an exception has occurred. If the user continues execution and there are still events pending, then the debugger acts as if the program were running and another event just occurred.

If the user interrupts execution of the program and there are no events pending, we stop all processors and determine their virtual time. The β time is the lowest virtual time of all the processors and no roll forward is necessary because there are no late operations. For either events or interrupts, we never stop the program with any late operations because the β time is always the lowest virtual time of all the processors.

4.3.4 Disallowing user commands

After the program has stopped, it may not be in a state that is consistent with execution of the β program. If it is not in a consistent state, then using the debugger to examine or modify the program state might lead to incorrect behavior of the debugger. To avoid incorrect behavior, the debugger must decide which commands to disallow. A command

must be disallowed if its behavior is changed by early operations. For example, an early write of a variable would cause the debugger to display the wrong value if the user examined that variable.

Computing the set of early operations is the first step in deciding if a command should be disallowed. We explain this in Section 4.3.4.1. Next, the debugger must determine the effect that the early operations have on the particular command. In Section 4.3.4.2 and Section 4.3.4.3, we explain what type of early operations will cause incorrect behavior for each of the debugger commands.

4.3.4.1 Computing the set of early operations

The set of early operations are the instances of statements that are executed that have a virtual time greater than or equal to the β time. There can be no general and practical method to compute the exact set of early operations because the execution of some operations can be data dependent. For example, if there were a conditional inside a loop, and the loop executes some early iterations, we would have to know which way the conditional branched in the past to know if the operations contained in the then clause are early operations. This would require the debugger to record a complete trace of execution of operations until it is certain that the operations are not early.

We can use a conservative approximation for the set of early operations. In this context, a conservative approximation includes all operations that are executed early and may include some operations which were not executed early. This may lead us to incorrectly disallow a command, but we will never permit a command that should have been disallowed.

We can compute the set of early operations for one processor by computing for each statement the set of possible virtual times over the entire execution of the program. We then eliminate all the virtual times that are less than the β time or greater than or equal to the current time of the processor.

Computing the set of virtual times for a statement is a straightforward extension of the method we use to compute the current virtual time of a processor, as is described in Section 4.2.1.2. For each statement, we start with the virtual time template. The values for `dist` variables can be determined from the processor index. To compute the set of virtual times possible, we initially assume that the value for all loop counters in a virtual time template have a range from 0 to ∞. Next we remove virtual times from the set that are less than the β time or greater than the current time.

For the example ω program in FIGURE 4–4, assume that processor 0 stopped at virtual time (2,3,6,2,6) and processor 1 stopped at virtual time (2,6,3,1,4). The β time would be (2,3,6,2,6). For processor 0, the β time is the same as the current time of the processor, so we know that there are no early operations. For processor 1, there are early operations. The first line of the ω program has the virtual time set of $(2,n,2,0,0)$ where $0 \le n \le \infty$. The first virtual time of the set that is greater than or equal to the β time is (2,4,2,0,0), so we can bound n from below by 4. The highest virtual time that is less than the current time is (2,6,2,0,0), so we can bound n from above by 6. This leaves $(2,n,2,0,0)$, $4 \le n \le 6$ as the set of early operations for the first statement.

The second line of the ω program is contained inside a `dist`. Its virtual time template contains the `dist` variable `a`, which has a value of 0 on processor 0 and 1 on processor 1. Because the virtual time template of the statement is different on each processor, the virtual time set of the statement on each processor is different. On processor 1, the set is $(2,n,3,1,4)$, where $0 \le n \le \infty$. The lowest virtual time greater than or equal to the β time is (2,4,3,1,4). The highest virtual time that is less than the current time is (2,5,3,1,4). This leaves $(2,n,3,1,4)$, $4 \le n \le 5$ as the set of early operations for the second statement.

For the third statement of the ω program, there are two loop counters. The set is $(2,n,6,m,6)$ where $0 \le n \le \infty$ and $0 \le m \le \infty$. For the low time, 3 and 2 are chosen as values of the `j` and `k` loop counters because that is their value in the β time. The low time is (2,3,6,2,6) and the high time is $(2,5,6,\infty,6)$. The set of early operations is $(2,3,6,m,6)$, $2 \le m \le \infty$ and $(2,n,6,m,6)$, $4 \le n \le 5$ and $0 \le m \le \infty$. The set of early operations for the fourth statement of the ω program is $(2,3,6,m,8)$, $2 \le m \le \infty$ and $(2,n,6,m,8)$, $4 \le n \le 5$ and $0 \le m \le \infty$.

FIGURE 4-4 β program and ω program

```
β program:
1            rep a = 0 to _np[0] with <a>
2  <0>       for (j = 0; j < n; j++) {
3                  dist a = 0 to _np[0] with <a>
4  <0>             b = 1;
5                  rep a = 0 to _np[0] with <a>
6  <0>             for (k = 0; k < m; k++)
7                     rep a = 0 to _np[0] with <a>
8                     A[k] = b;
9            }
ω program:
2,j,2,0,0    for (j = 0; j < n; j++) {
2,j,3,a,4          b = 1;
2,j,6,k,6          for (k = 0; k < 10; k++)
2,j,6,k,8          A[k] = b;
             }
```

4.3.4.2 Examining and modifying variables

If we allow the user to examine or modify a variable when there are early or late operations, we must observe the same dependence rules that a compiler must observe when scheduling code. These rules determine when operations may be reordered. We use the terminology from data dependence[Padua 86] to phrase the constraints.

There is a *flow dependence* between two operations when one operation stores to a variable and a later operation reads it, without any intervening stores. In the following example, the second statement is flow dependent on the first statement because the first statement stores into `a` and the second statement reads `a`.

```
a = 1;
b = a;
```

There is an *output dependence* between two operations when they store to the same variable and there are no intervening stores. In the following example, there is an output dependence between the first and second statement because they both store to the same variable.

```
b = 1;
b = 2;
```

There is an *anti dependence* between two operations when one operation reads a variable and a later one writes the same variable.

If the user wants to examine a variable, then there can be no late or early writes. Examining a variable is treated the same as a read of the variable at the point in the program corresponding to the β time. If there are late writes of the examined variable, then allowing the user to look at its value would be the same as moving a read before a write, which violates a flow dependence. If there is an early write, then we would be moving a read after a write, which violates an anti dependence.

If the user wants to modify a variable, there can be no late or early operations which read or write the variable. A late reads violates an anti dependence, a late write violates an output dependence, an early read violates a flow dependence, and an early write violates an output dependence.

In Section 4.3.4.1, we compute the set of operations that are early; we can use this information to determine which variables have early reads or writes. If a statement has an early execution, then the variables in that statement have early reads or writes. Since the early execution information is conservative, the early read and write information is conservative as well.

In the example in FIGURE 4-4, we concluded that processor 0 has no early operations, thus it doesn't have any early reads or writes, and all variables on that processor can be examined and modified at the current point in the execution of the program. Processor 1 has executed some early operations. The first statement has been executed early, so we can conclude that the variables `j` and `n` have early reads and the variable `j` has an early write. Because the second statement has been executed early, we can conclude that the variable `b` has some early writes. The third statement has also been executed early, so we can conclude that the variables `k` and `b` have early reads and the variable `k` and the array `A` have some early writes. The only variable that doesn't have an early write is `n` on processor 1, so that is the only variable that can be examined by the user. Every variable on processor 1 has an early read, so none of them can be modified.

We have conservatively assumed that if a statement has an early read or write of an array, then the entire array has an early read or write. If we have more information about which element of an array each iteration of the loop references, we can compute finer grain information about which elements of the array have early reads or writes. In the example program, we know that iteration *i* of the loop in the third statement writes element `A[i]`. If the current time of processor 1 is (2,3,6,4,6) and the time of processor 0 is

still (2,3,6,2,6) as it was in the previous example, then the β time is still (2,3,6,2,6). The only early operations are of processor 1 on statement 3, which is (2,3,6,n,6) where $2 \le n \le 3$. In this case, there are early writes of the array elements A[n] where $2 \le n \le 3$.

4.3.4.3 Setting breakpoints

If a breakpoint is set on a statement that has early executions, then some of the breakpoints that the program would have normally reached will be missed. The debugger cannot be certain because the set of early operations is conservative; we only know that the statement might have been executed early. If the user sets a breakpoint on a statement with early executions, then the debugger should warn the user that some breakpoints might have been missed. The debugger has enough information to tell the user which iterations the missed breakpoints come from.

4.3.5 What to do when commands are disallowed

When a command is disallowed because some processors have executed early operations, the user might be able to get the information that they need another way. They might inspect another variable or set a breakpoint on another statement. However, in some cases, they may have to inspect a particular variable. In this situation, they could try to run their program again and hope that the next time it stops in a state that has the information they need. Even if we run the program again, it might not stop in a better state on successive tries.

There are several ways that a debugger can help this problem. It can rerun the program from the beginning and stop in a state with the same β time, but no early operations. The debugger could also restrict execution of the parallel program so that when it stops at a breakpoint, it will be in a state without any early operations. We call this a *consistent breakpoint* because the program stops in a state that is consistent with a state in the execution of the β program. Consistent breakpoints can sacrifice some or all of the parallelism of the program but might cost less in the long run when compared to repeatedly rerunning the program.

Both of these choices can be viewed as a special case of roll forward. The first is rolling forward all the processors from the beginning of the program and stopping at a specified virtual time, which in this case happens to be the current β time. The second is rolling forward all processors to the virtual time of the next potential breakpoint.

We first introduce the mechanisms that are needed for roll forward in Section 4.3.5.1. We then explain how roll forward can be used to do rerun in Section 4.3.5.2 and consistent breakpoints in Section 4.3.5.3.

4.3.5.1 Roll forward

For roll forward, we want to advance the execution of all processors until they have executed all statements with a virtual time less than some *target* time t. We shall call the statement in the β program whose virtual time is the same as the target time the *target statement*. To roll forward, we could just single step every processor and check the virtual time after every step, stopping when the current time of a processor is greater than or equal to t. However, this is very costly. Instead we would like to compute in advance

the statement that each processor should stop at, and set a breakpoint there. If that statement is inside a loop, then the first time that the statement executes might not be the time that we want to stop. The breakpoint must be conditioned on the virtual time being greater than or equal to the target time. We shall call the statement and the virtual time that a processor should stop at the least upper bound (LUB) because it is the lowest virtual time of the processor that is greater than or equal to the target time. For any given target time, each processor will have its own LUB.

We can compute the LUB for each processor by starting at the target statement in the β program and tracing the control flow until we find one statement and virtual time for each processor. Every time we find a PIL for a processor that does not yet have a LUB, we record the virtual time and statement as the LUB for that processor, and then remove the processor from the list of processors that do not have LUB's. This method of finding LUB's assumes that the program executes the statement at the target time. If the control flow never gets to the target statement, there is no guarantee that it will get to the LUB either. The previous method that single steps until we reach the target time always works.

The program in FIGURE 4-5 illustrates this procedure. The program at the top is a β program and the program below it is the same β program with the `rep` and `dist` constructs expanded. All of the following discussion refers to the expanded β program.

If the target time is (2,3,5,1,6), then we start the search for the LUB at the second assignment to the variable `a` because only this statement can have the target time as a virtual time. The statement is mapped to processor 1, so this is the LUB for processor 1. If we trace the execution of the program, we skip over the `else` clause and execute the next `if` statement. The `if` statement is mapped to both processors, so this statement is the LUB for processor 0. The virtual time of the LUB is in the same iteration as the target time, (2,3,12,0,0). When tracing control flow, we never have to trace the body of an `if` statement after checking the part of the statement that contains the condition. Recall that the PIL's of the condition line of an `if` statement must be a superset of the PIL's of the statements in the body. If we don't get a match on the PIL of the line with the condition, we cannot get a match on any PIL in the body. This is important because we cannot predict future behavior of the program, and cannot know whether to trace the then or else clause when searching for the LUB.

For this example, when we want to roll forward to a target time of (2,3,5,1,6), the debugger sets a breakpoint on line 6 on processor 1. When the value of the loop counter `i` is 3 when executing this statement, the debugger stops the processor. For processor 0, the debugger sets a breakpoint on line 12, and stops the processor when it reaches the breakpoint and the value of `i` is 3.

Another example illustrates how to trace control with loops. If the target virtual time is (2,3,16,1,17), then the starting point in the program is the last statement in the loop. This statement is mapped to processor 1, so this is the LUB for that processor. If we trace the execution of the program, we follow the loop back path to the beginning of the loop. The loop header is mapped to processor 0, so this statement is the LUB for that processor. The virtual time of the LUB is in the next iteration of the loop, (2,4.0,0,0). Just like the `if`, we never have to trace through the body of a loop after checking the header because the PIL's of the header are a superset of the PIL's of the body. If we do not get a

FIGURE 4-5 A β program and its expanded version.

β program

Virtual time			
	1		rep for m = 0 to 1 with <m>
(2,i,0,0,0)	2	<0>	for (i = 0; i < 10; i++) {
	3		rep for m = 0 to 1 with <m>
(2,i,3,0,0)	4	<0>	if (a == 1) {
	5		dist for n = 0 to 1 with <n>
(2,i,5,n,6)	6	<0>	a = 2;
	7		} else {
	8		dist for n = 0 to 1 with <n>
(2,i,8,n,9)	9	<0>	b = 2;
	10		}
	11		rep for m = 0 to 1 with <m>
(2,i,12,0,0)	12	<0>	if (e == 1) {
	13		dist for n = 0 to 1 with <n>
(2,i,13,n,14)	14	<0>	c = 2;
	15		} else {
	16		dist for n = 0 to 1 with <n>
(2,i,16,n,17)	17	<0>	d = 2;
	18		}
	19		}

β program with expanded `rep` and `dist`

Virtual time		
(2,i,0,0,0)	<0>,<1>	for (i = 0; i < 10; i++) {
(2,i,3,0,0)	<0>,<1>	if (a == 1) {
(2,i,5,0,6)	<0>	a = 2;
(2,i,5,1,6)	<1>	a = 2;
		} else {
(2,i,8,0,9)	<0>	b = 2;
(2,i,8,1,9)	<1>	b = 2;
		}
(2,i,12,0,0)	<0>,<1>	if (e == 1) {
(2,i,13,0,14)	<0>	c = 2;
(2,i,13,1,14)	<1>	c = 2;
		} else {
(2,i,16,0,17)	<0>	d = 2;
(2,i,16,1,17)	<1>	d = 2;
		}
		}

match on any PIL of the header, we cannot get a match on any PIL in the body. This is important because we cannot predict the future behavior of the loop, other than to know that if the loop executes the body, it will always execute the header one more time to check if it should execute the next iteration.

4.3.5.2 Rerun

For rerun, we want to put the program in state where it has the same β time as the current state, but there are no early operations. We can do this by starting the program from the beginning and rolling forward to the β time.

When we rerun a program, the final state should be the same as the original state, except that there should not be any early operations. The state is the same if the program is determinate and the execution environment is reproduced. Code produced by thread splitting is determinate, thus there cannot be any schedule dependent bugs that prevent rerun from reaching the desired state. To reproduce the execution environment, external I/O must be the same, memory should be initialized the same way, etc. Reproducing the execution environment can be difficult. However, users typically design their programs so that they can be rerun because debugging a program usually requires that the program be run many times. A debugger can make this easier by initializing memory and logging and playing back any external I/O[Pan 88].

4.3.5.3 Consistent breakpoints

When users set breakpoints, they are usually interested in the state of the program at that point in the execution. Rather then letting the program run unrestricted, we can use a consistent breakpoint, where the debugger controls execution so that the program breaks in a state that is consistent with a state of the β program. This can reduce the parallelism of a program. We set breakpoints without specifying the iteration; if the breakpoint is set on a statement inside a conditional, the program could stop at any iteration. If we want to be sure that we have not executed any early iterations, then we must execute one iteration at a time.

A consistent breakpoint can be implemented with roll forward. For each breakpoint set by the user, we compute the time of the next breakpoint. This can be done by finding the smallest virtual time in the set of virtual times for a breakpoint statement that is greater than or equal to the β time. For each processor, we must check if there are any early operations, assuming that the time of the earliest breakpoint is the next β time. If there are early operations for that β time, then we cannot have a consistent breakpoint. Next, for every processor we compute one LUB for each breakpoint and roll forward to the target time.

If the breakpoint is in a conditional or a loop, then the breakpoint may or may not occur, depending on whether the program executes the body of the conditional or loop. If the breakpoint does occur, then the roll forward mechanism leaves the program in a state where there are no early operations. If the breakpoint does not occur, then the other processors may or may not stop at the LUB for the breakpoint since we only guarantee that a processor will reach a LUB if the program reaches the target statement.

If a processor stops at the LUB for a breakpoint, and the breakpoint does not occur, then we must restart the processor. The difficulty is knowing when it is safe to restart. If any processor runs past the target virtual time then the breakpoint will not occur. If any pro-

cessors are stopped at the LUB and other processors are still running we must poll the running processors for their virtual time until they all stop at the LUB or one of the processors runs past their LUB. If a processor runs past its LUB, we start over. The debugger stops all processors, computes the earliest virtual time for each breakpoint and the new LUB's, and continues the roll forward.

If we need to break at every iteration of a parallel loop, then there is little parallelism that can be exploited in that part of the program. However, there may be parallelism earlier in the program. Consistent breakpoints allow the user to take advantage of parallelism in parts of the program that the user does not care about, while avoiding the need for rerun. In the worst case, using consistent breakpoints is comparable to executing with a sequential schedule.

4.4 Efficiency

A debugger with dynamic order restoration must do extra work when compared to a conventional debugger for sequential programs; however, we believe that the additional work is small enough to make dynamic order restoration practical. In this section we examine why there is extra work and estimate the cost. There are two main sources of additional work for the debugger when compared to a debugger for sequential programs: the information that must be computed to decide if commands are disallowed and re-execution when commands are disallowed. We discuss each of them below.

During normal execution, no additional work is necessary. When a program stops executing because of a breakpoint, exception, or interrupt, the debugger first computes the virtual time of each processor to determine the current time. Determining the virtual time takes a constant cost per processor. After the current time is set, the debugger rolls forward all processors with late operations. The debugger must compute the LUB for each processor when rolling forward. Computing the LUB of a processor is linear in the size of the program. Rolling forward takes time, but since useful work is being executed, we do not consider this extra. After roll forward is complete, the debugger computes the set of early operations on each processor, which is linear in the program length. From the set of early operations, the debugger can compute the set of variables with early reads and writes, which is also linear in the program length. Computing early reads and writes with a resolution of individual array elements can require some data flow analysis. However, this information can be precomputed at compile-time. All the work described above must be done every time the program stops and is linear in program length and the number of processors. If either the program size or the number of processors is so large that the time is significant, we could use the parallel processor to compute the information; the problem can be partitioned by dividing the program or dividing by processor.

If a command is disallowed, the user may decide to re-execute the program. If the program is re-executed frequently, the cost can be significant; in some cases, it could be better to execute the program with a sequential schedule so that commands are never disallowed. The number of times it is necessary to re-execute a program depends on what variables are examined and modified and where breakpoints are set, and is thus dependent on the user. We believe that the expense of re-execution will in general be

small, especially when it is compared to the cost of executing with a sequential schedule.

First, if a program has a factor of n speedup when executing on the parallel machine, the debugger would have to rerun the program n times before execution would be as slow as sequential execution. For massively parallel machines, n can be quite high.

Second, after re-executing the program once, the program is in a state without early operations, so no commands are disallowed in that state and re-execution is not needed again until the user continues execution.

Third, the user can use consistent breakpoints to ensure that re-execution is never necessary. As explained in the previous section, if we set a consistent breakpoint in a parallel loop the program executes sequentially, so any breakpoint, exception or interrupt would put the program in a state without any early operations. In the worst case, the user can use consistent breakpoints to execute the program with a sequential schedule.

4.5 Related work

Gupta studies the problem of providing source level debugging for programs that have been trace scheduled for VLIW (Very Long Instruction Word) machines [Gupta 88]. Since a VLIW can execute many operations in the same instruction word, it can be viewed as a tightly synchronized parallel machine. Our work focuses on the dynamic nature of execution on MIMD machines. In a MIMD machine, the processors are not tightly coupled and the relative order of execution of two operations on different processors can change, which creates the need for dynamic order restoration. It is unnecessary for a VLIW because the parallelism is statically scheduled. Gupta's work focused on static reordering of operations, which we do not consider in this thesis. However, we can apply his results to the problem of structural mapping. For this reason, Gupta's work on the structural problem is complementary.

Pineo and Soffa study source-level debugging for automatically parallelized code [Pineo 91]. They only attempt to solve the problem of presenting the correct data values to the user; they do not consider modifying data or control flow.

Their method exploits the features of single assignment languages. Over the lifetime of a program, a variable will have many versions, one version for each time it is assigned a value. They make the versions explicit in the program by converting the imperative, sequential program to a sequential program in single assignment form. Every time a variable is assigned a value in the imperative program, a new variable with a unique name is created in the single assignment program. If a variable is assigned a value in a loop, scalar expansion is applied so that every iteration will assign a different variable. If the bounds of a loop are not known before the loop begins, then space must be incrementally allocated for variables during the execution of the loop. Each position in the source program has associated with it the versions of all the variables that are live at that point. Only one version of each variable can be live at one point in the program. After this conversion, the program is parallelized.

When debugging with our method, the debugger must determine if the current value of the variable is the correct one to inspect for the current β time. In their method, every variable is only assigned one value, so they must determine which copy of the variable is the correct one for the current point in the program. To determine which copy, they use the program counter in the source program and the value of loop counters in the source program. In their system, they do not present the user with a model that only a single loop iteration is being executed at one time, so the user must provide the loop counter values when examining variables (e.g. examine variable a from iteration 5).

As Pineo points out, the single assignment transformation is not sufficient to solve the problem on MIMD machines. Even if a variable can only be assigned to once, it still has two values during the lifetime of a program: uninitialized and initialized. After the first version of the variable in the source program has been initialized, it is still possible that the copy that we want to inspect has not yet been initialized. To solve this problem, Pineo proposes initializing all memory locations to 0 and when the debugger is asked to display a variable that happens to be 0, the debugger reports that the value is 0 or the value is not ready yet (the debugger does not have enough information to distinguish the two cases). Alternatively, if the hardware supports detection of reading uninitialized variables, this can be checked automatically. Our virtual time scheme eliminates the need for special hardware and still can give precise answers about the values of variables.

There are four key differences between Pineo's and our work. First, we provide control flow that is consistent with the sequential program. Not providing source program control flow places a greater burden on the user of understanding the parallel structure of the program.

The second difference is that Pineo's method requires extra space for the storage of data. a large change in the parallel code. The conversion to single assignment form can require significant increases in space and computation solely to support debugging. To reduce the space requirements, Pineo performs name reclamation to re-use storage for multiple versions of the same variable when it isn't necessary to keep all the versions around for debugging or parallelization. In benchmark programs, Pineo measures a 10% increase in storage that can be attributed solely to debugging; however, the number can be much higher for individual programs.

The third difference is that saving multiple versions of a variable ensures that a current value of a variable can be found if we run the program long enough so that the value is initialized. In our method, the current value might have been overwritten and can only be found by rerunning the program. The single assignment method pessimistically uses extra space and computation in case the user might need to see a value. We believe that it is more effective to not do the extra work, and rerun the program when it is not possible to provide a current value of a variable because it has been overwritten.

The fourth difference is that the single assignment method can support reordering of stores on the same processor, while our method cannot. As described earlier, this capability comes at a potentially large cost, and the user must pay the cost even when there is no reordering of stores. However, saving extra copies of variables to achieve the same result could be added to our model if desired.

4.6 Summary

In this chapter, we define the thread splitting transformation and its D function. We show that thread splitting is debuggable if a sequential schedule is used. In a sequential schedule, we execute the statements of the ω program in the same order as they are executed by the β program. This ordering is determined by an assignment of virtual times to statements. When we allow the program to execute in parallel, we may stop in a state that is not consistent with a state of the β program. If we allow the user to examine or modify variables in this state, the behavior of the debugger might be correct. The debugger must detect if allowing the user to modify or examine or variable will cause incorrect behavior.

CHAPTER 5

Distribution transformations

In the distribution phase of compilation, code and data are assigned to processors for execution. Distribution is accomplished in different ways on different machines. On shared memory machines, the parallel DO can be implemented with fork/join parallelism, where a master thread forks off other threads which each execute a subset of the iterations[Stevens 90]. After all the iterations have been executed the threads join, so that only the master thread continues execution. Compilers for distributed memory machines usually use a different model, called SPMD, where the same program is executed on all processors. Every processor executes the sequential code and when the parallel loop is executed, every processor executes its assigned iterations [Tseng 89]. Synchronization is not done after every parallel loop; it is only done on an as-needed basis. The thread splitting model was designed to support SPMD programs. The rest of this chapter will assume SPMD type loop distribution.

The general problem of deriving the β program for a particular distribution can be more difficult than generating the parallel code itself. The β program (the β program is the step before thread splitting) is essentially the parallel program combined into a single thread of control. The order of operations in this thread of control must be compatible with the order of operations in both the ω program and the α program. Furthermore, the communication in the single thread must be ordered so that it will not deadlock (receives must follow sends).

In this chapter, we describe how to derive the β program for a limited domain taken from loop based parallelism. From a description of the distribution of the program we can automatically derive the β code and its associated D function. To give some context and demonstrate the relevance of the domain we have chosen, we briefly describe how a compiler that takes advantage of loop-based parallelism can translate a loop nest into a parallel program in Section 5.1. We describe the domain of source programs in Section 5.2. In Section 5.3, we introduce the basic method for distributing iterations among pro-

cessors. This is sufficient to do the block mappings employed by most compilers. Section 5.4 describes how communication can be inserted into the program, and Section 5.5 presents the basic method for distributing data. In Section 5.6, we extend the basic method for distribution of iterations and data so that we can support cyclic and block-cyclic distributions of iterations and data as well.

5.1 Parallelizing compilers

For distributed memory machines, compilers usually distribute the data and then use the location of the data to determine where to perform the computation. One commonly used strategy is the owner computes rule, which assigns the computation to the processor that owns the variable where the result is to be stored. In this case, the distribution of iterations matches the distribution of data.

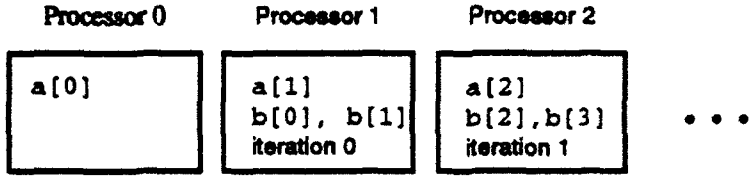
Two important factors for determining how to distribute the data are locality and load balancing. A program has good locality when the data items that a processor needs are usually on that processor. If the processor must execute a statement that needs a value from another processor, then that value must be fetched by the processor The better the locality, the less time spent shipping data around, which leads to good performance. The other factor is load balancing; if the work of a program is not evenly distributed, then some processors will be under-utilized and the performance will be poor. Locality and load balancing often place conflicting constraints on the mapping of data and computation. If all the data and computation were placed on one processor, then no communication would be necessary at run time. However, because one processor is doing all the work, the load balancing is not good. The more computation is distributed, the better the load balancing, but the locality becomes worse which leads to more communication.

Nested loops with regular access patterns are good candidates for parallelization for distributed memory machines because the compiler can trade off locality and load balancing at compile time. Algorithms for determining the best distribution are beyond the scope of this thesis, but have been studied extensively in the literature[Gupta 92][Wholey 91][Li 91][Knobe 90].

The example in FIGURE 5-1 illustrates some of the constraints for deciding how to distribute data and computation and the different ways for mapping data to processors. In the loop, iteration 0 stores its result into `a[1]`, so if `a[1]` resides on processor 1, then we want to execute iteration 0 on processor 1. Iteration 0 also accesses array elements `b[0]` and `b[1]`, so we want to put those elements on processor 1 as well. Essentially, when we distribute the array `b` across the processor array, we shift it to the right by one and assign two elements per processor. Assigning location based on locality is called alignment. In this example the array `b` is aligned to the array `a` so that no run time communication is necessary to execute an iteration of the loop. The domain examined in this chapter allows us to express mappings similar to those necessary for the example in FIGURE 5-1.

FIGURE 5-1 The placement of data and iterations for a loop

```
for (i = 0; i < 10; i++)
   a[i+1] = b[2*i] + b[2*i+1];
```



5.2 Domain description

In our domain, source programs are limited to nested loops. The iterations of a loop may be distributed or replicated across a set of processors or all executed on the same processor. We will use the C syntax of `for` loops. However, loops where iterations are distributed must be of the following form:

`for (`*counter* `=` *low*`;` *counter* `<` *high*`;` *counter* `+=` *step*`)`

For loop control *low*, *high*, and *step* may be expressions but may not change value once the loop has begun. The body of the loop may contain any statements in the language, including conditionals and loops, but may not change the loop counter. This is essentially the semantic equivalent of a FORTRAN DO loop.

To specify the parallelizing transformation, the compiler must determine the mapping of loop iterations to processors, the description of inter-processor communication, and the decomposition of data. To simplify the presentation, the generation of β code will be presented as a three step process: distributing iterations, inserting communication, and distributing data.

The first step, iteration distribution, restructures the loop and adds processor index labels so that the appropriate loop iterations will be performed on each processor. Communication statements are inserted into the loop in the second step. Finally, global names and addresses for distributed data are replaced with local names in the third step.

Code generation and debugging procedures are described for each step. The final output will be the β program and the debugging function for the transformation between the α program and the β program.

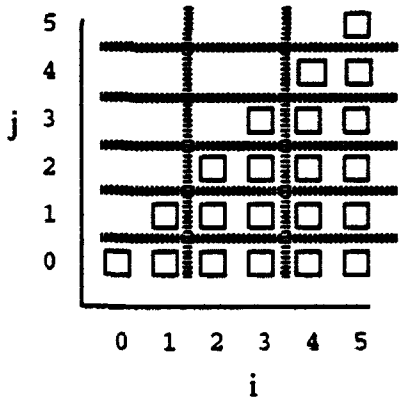
5.3 Basic loop iteration distribution

In this section, we introduce the basic method for distributing loop iterations. More complicated distributions build upon this model.

The distribution of loop iterations to processors will be described as a mapping between elements of the iteration space and the processor space. The mapping is determined by the compiler. An *iteration space* for a q deep loop nest is the q dimensional space N^q. Each point in the space corresponds to one loop iteration; the value of the loop counters for that iteration are the coordinates of the point. FIGURE 5-2 shows a doubly nested loop and its corresponding 2 dimensional iteration space. The *processor space* for a r dimensional processor array is the r dimensional space N^r. Each point corresponds to one processor. The mapping assigns a set of iterations to be executed on each processor.

FIGURE 5-2 Iteration space of a doubly nested loop

```
for (i = 0; i < 6; i++) {
   for (j = 0; j < i; j++) {
      ...
   }
}
```



Each processor executes its assigned iterations in the same order that they were executed in the original program.

We only consider a limited class of linear mappings. A linear mapping from a q dimensional iteration space to a r dimensional processor space can be described with T, a $r \times q$ matrix and o, a r element vector. The T matrix and o vector are chosen by the compiler which decides the mapping. The mapping is computed by:

Equation 5.1 $$p = \lfloor T(i) + o \rfloor$$

where i is a point in the iteration space (*iteration index*) and the result p is a location in the processor space (*processor index*). Since processor indexes are composed of integers, the result of $T(i) + o$ is always rounded down (floor function). Mappings that result in a processor index that does not correspond to an actual processor index are invalid and cannot occur if the compiler is correct.

Mappings will further be restricted so that in every row and column of T at most one element is non-zero. In effect, the iteration space is divided into slices that are parallel to one axis of the iteration space. These slices are distributed along a dimension of the processor space. The dotted lines in FIGURE 5-2 show one possible way of slicing the iteration space. Our notation permits more complicated linear mappings, for example mappings that slice the iteration space with cuts that are not parallel to the axes. Supporting more general linear mappings does not make debugging more difficult, but complicates the generation of loop bounds for distributed loops; it is not considered in this thesis.

When it is necessary to map the same loop iteration to more than one processor, an element of T may also be the special symbol *. Arithmetic expressions that contain * will be evaluated as follows: $\forall (c \neq 0) : * \times (c) = *, * \times (0) = 0, \frac{*}{c} = *,$ $* \bmod n = *$, and $* + c = *$. If a component of a processor index is computed to be *, then that component takes on all possible values for that dimension of the array. For example, in a 2x5 array, the processor index $\begin{bmatrix} * \\ 2 \end{bmatrix}$ represents the processor indexes $\begin{bmatrix} 0 \\ 2 \end{bmatrix}$, and $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$.

The mapping described above is sufficiently general to do the block distribution, distribution by row, and distribution by column, which is used by most parallelizing compilers. It does not include the cyclic mapping because it is non-linear. In Section 5.6, we extend the basic iteration distribution to handle cyclic distributions.

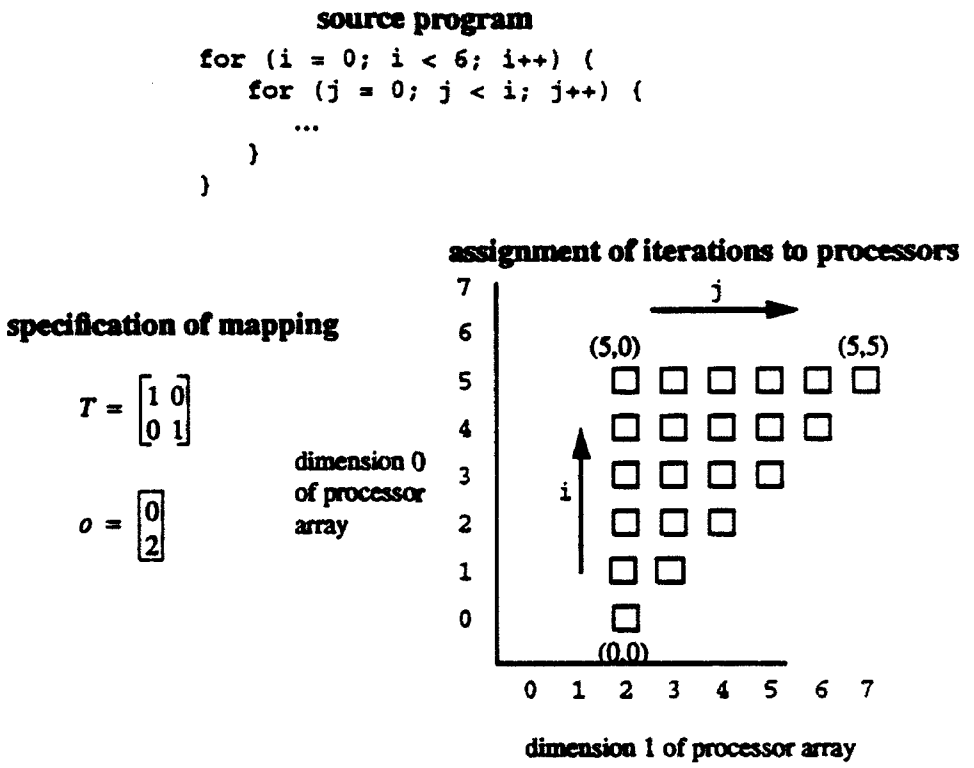
5.3.1 Examples

In this section we will give examples of various iteration mappings and their transformation matrices.

The first example is a mapping where the offset vector o is non-zero. The mapping in FIGURE 5-3 converts an iteration index into a processor index by adding $\begin{bmatrix} 0 \\ 2 \end{bmatrix}$. Each square represents one iteration, the coordinates of the iteration is the processor index that executes the iteration. We have labeled the iterations on the corners with their iteration indexes.

In this example, a row of processors executes one entire iteration of the outer loop. Iterations of the inner loop are spread across each row. If we swap the columns of the map-

FIGURE 5-3 Iteration mapping with an offset



ping matrix, then we would obtain a mapping that is the transpose of the one above; columns execute outer loop iterations and inner loops are spread across each column.

Sometimes it is desirable to map all the iterations of a loop to a single processor. The mapping matrix in FIGURE 5-4 assigns each iteration of the outermost loop to a different processor, but does not distribute the innermost loop.

If a loop is distributed across a dimension, but more than one iteration is mapped to the same processor, than a matrix element with an absolute value less than one 1 should be used for the appropriate dimension. The matrix in FIGURE 5-5 maps 2 iterations of the outer loop to each row of processors, and spreads the iterations of the inner loop so that each processor gets 3 iterations of the inner loop for a total of 6 iterations. Processors near the boundary of the iteration space will get less iterations. Using a coefficient greater than 1 will spread iterations so that not every processor will have work to do. For example, if the value 3 is used, then only every third processor will execute an iteration.

When a negative coefficient is used, the iterations with successively increasing loop counter values can be mapped to processors with successively decreasing processor

FIGURE 5-4 Iteration mapping that puts a dimension on the same processor

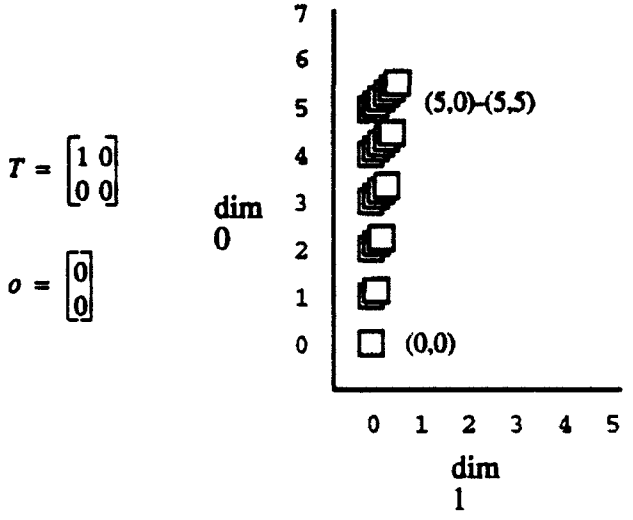
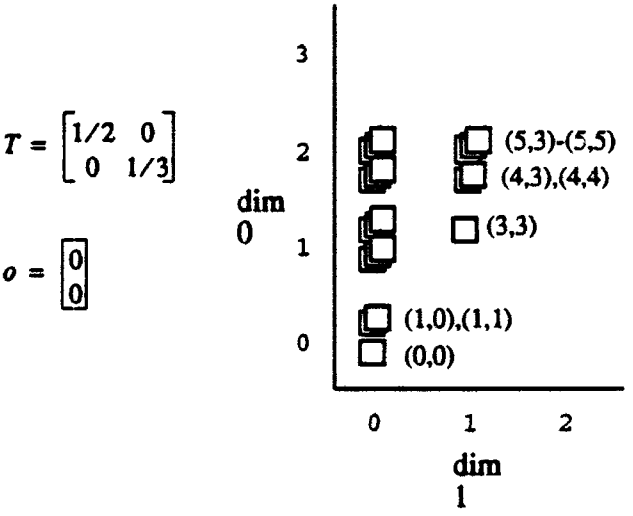
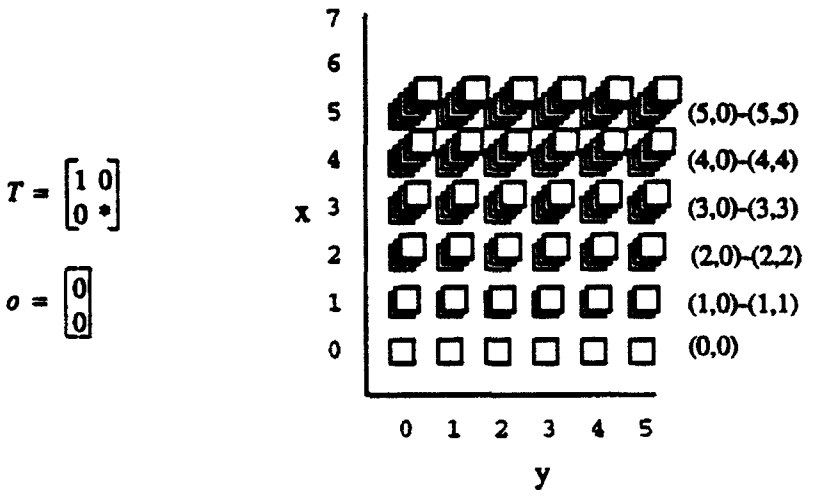


FIGURE 5-5 Mapping that places more than one iteration on the same processor



indexes. A * in T will result in a processor index that contains a *. In the example of FIGURE 5-6, the outer loop is distributed by row and every column in the row executes the entire inner loop for that row.

FIGURE 5-6 Mapping where iterations are duplicated on several processors



5.3.2 Code generation

The code generation problem is to take a mapping and a loop nest and generate a β program with a processor index labelling. The labelling should be chosen so that when thread splitting is applied, the output will be a parallel program where each processor executes the appropriate set of iterations. Furthermore, the order of execution of iterations in the β program must be the same as it was in the original program. If the compiler wants to change the order of execution of iterations on a single processor, then that can be done as a part of the restructuring phase.

Each row of the T matrix determines how iterations are mapped to one dimension of the processor array. If an element in column c of row r is neither a zero nor a *, then the iterations of the cth loop in the nest are distributed across dimension r of the processor array. If all the elements of row r are 0, then loop iterations are not distributed across dimension r of the processor array. If an element in the row is a *, then iterations are replicated on that dimension of the processor array.

Algorithm 5.1 takes a mapping and source code and outputs the β program. The algorithm is divided into 4 phases, as marked. Phase 1 analyzes the T matrix to decide which loops are distributed, replicated, and not distributed.

Phase 2 handles the case when work is not distributed across a dimension of the processor space. An example is when iterations are only assigned to one column of processors in a 2 dimensional array; work is not distributed across the columns of the processor space. We use conditionals to prevent the processors that are not assigned any iterations from executing. We first emit a `dist` to distribute copies of the code across the dimension. Then an `if` statement is used to restrict execution to the appropriate processors.

Phase 3 of the algorithm emits each loop of the nest. If that loop is distributed, a `dist` is emitted that distributes the loop code across the appropriate dimension. Iterations are distributed from low values to high values if there is a positive value in *loopscale* and are distributed from high values to low values if there is a negative value in *loopscale*. For some programs, it is not known at compile-time whether the loop should go from low to high or high to low. An example is where the sign of the step for a loop is unknown at compile-time. If the sign is not known, then we must emit two loops, one that distributes from low to high and another that distributes from high to low. A conditional selects which copy should be executed at run-time. The restriction that PIL's be constant forces us to emit two loops for these situations. The reason for constant PIL's and the effect of their limitations are described in CHAPTER 7. For the rest of this chapter, we assume that we know at compile-time whether loops should be distributed from low-valued processors to high-valued processors or vice versa.

If the compiler emits a loop in phase 3 that is distributed, the bounds of the loop are parameterized by processor index so that each instance created by the `dist` will execute its assigned loop iterations. If the loop is not distributed, it is emitted unchanged.

Phase 4 of the algorithm emits the loop body. For every dimension of the processor for which iterations are replicated, we emit a `rep`. Then we emit the body itself, with a PIL of 0.

The auxiliary functions comp_slice_begin and comp_slice_end are used to compute the first and last iterations that a processor should execute in a distributed loop.

Algorithm 5.1

Generate β program for a mapping matrix and offset.
Input:
depth - depth of loop nest
dim - dimension of processor space
T[*dim*][*depth*] - mapping matrix
o[*dim*] - offset vector
counter[*depth*] - names of loop counters of nest, where *counter*[0] is outermost loop
upper[*depth*] - upper bound expressions for loop nest
lower[*depth*] - lower bound expressions for loop nest
step[*depth*] - step size expressions for loop nest
body - body of loop nest
Output: β loop nest
Intermediate storage:
looptype[*depth*] - the type of loop, dist if the loop is distributed, `rep` is the loop is replicated, and ndist if it is neither
looppdim[*depth*] - dimension of processor array that the loop is distributed over
loopscale[*depth*] - scale factor if loop is distributed
DIST - iterations are distributed across dimension *d* if $d \in DIST$
REP - iterations are replicated across dimension *d* if $d \in REP$
NDIST - iterations are not distributed across dimension *d* if $d \in NDIST$

Method:
phase1:
NDIST = *REP* = *DIST* = Ø;
foreach row *r* of *T* **do**
    if *r* is all zero's **then**
        NDIST = *NDIST* ∪ *r*;
    endif
    if element *c* of row *r* is a '*' **then**
        REP = *REP* ∪ *r*;
        looppdim[c] = *r*;
        looptype[c] = rep;
    endif
    if element c of row r is a number n **then**
        DIST = *DIST* ∪ *r*;
        looppdim[c] = *r*;
        loopscale[c] = *n*;
        looptype[c] = dist;
    endif
enddo

phase2:
/* for dimensions of the processor array that don't have distributed or replicated loops,
    emit a **dist** to distribute everything across the dimension, and then emit an **if** so
    that only the correct processors execute work */
if NDIST ≠ Ø **then**
    foreach *d* ∈ *NDIST*
        emit a `dist` for dimension *d*
    foreach *d* ∈ *DIST* ∪ *REP*
        emit `rep` for dimension *d*
    /* build a condition string, with one condition per member of *NDIST* */
    foreach *d* ∈ *NDIST*
        condition = *condition* ∪ "and *_id[d]* == *o[d]*";
    emit if statement with PIL of 0 and *condition* as the condition
endif

phase3:
/* each iteration emits one loop of nest */
for *l* = 0 to *depth*-1 {
    /* emit the `dist` if the loop is distributed */
    if *looptype[l]* == dist **then**
        emit `dist` that distributes its body across dimension *looppdim[l]*
        DIST = *DIST* - *looppdim[l]*;
    endif
    /* for all the dimensions of the processor array that don't have dists yet,
        emit a `rep` */
    foreach *d* ∈ *REP* ∪ *DIST*
        emit a `rep` for dimension d;

```
    /* now emit the loop header */
    if looptype[l] == dist then
        emit loop with PIL of 0
        loop lower bound is:
            comp_slice_begin(_id[looppdim[l]], lower[l],loopscale[l],offset[l]);
        new upper bound is:
            comp_slice_end(_id[looppdim[l]], upper[l],loopscale[l],offset[l]);
        step size is unchanged
    elseif loop is not distributed, emit it unchanged;
}

phase4:
/* loops are done, now do the body */
foreach statement s ∈ body
    foreach d ∈ REP
        emit a rep for dimension d;
    emit s with PIL of 0
}
```

End of Algorithm

Auxiliary functions:

```
comp_slice_begin(slice,lower,c,offset):
    max(lower,(slice-offset)/c)
comp_slice_end(slice,upper,c,offset):
    min(upper,(slice+1-offset)/c-1)
```

5.3.2.1 Examples

To illustrate how the algorithm works, we use three examples, one for each of the types of loops: distributed, replicated, and not distributed. All of the examples will share the same α program, which is the two deep loop nest found at the top of FIGURE 5-7. The program is mapped to a 2 dimensional processor array.

In part (a) of FIGURE 5-7, we start with a loop which is neither distributed nor replicated. All the iterations are executed on a single processor, $\begin{bmatrix}2\\1\end{bmatrix}$. After phase 1 of Algorithm 5.1 NDIST contains 0 and 1 while REP and DIST are empty. In phase 2, we emit a `dist` for dimensions 0 and 1, but don't emit any `rep`'s. We then emit an `if` with conditions for the two dimensions contained in *NDIST*. These conditions restrict the rest of the program to executing on a single processor. In phase 3, when we emit the loop, there are no members in *DIST* or *REP* so we don't emit any `rep`'s or more `dist`'s. The loop and body are emitted unchanged.

In part (b) of FIGURE 5-7, we have an example of a loop where the outer and inner loops are distributed. After phase1, *DIST* has dimensions 0 and 1 in it and *NDIST* and *REP* are empty. Because *NDIST* is empty, we can skip phase 2. For phase 3, we first emit a `dist` for dimension 0, then remove 0 from *DIST*. We then emit a `rep` for dimen-

FIGURE 5-7 Code generation examples for Algorithm 5.1

α program

```
for (i = m; i < n; i++)
   for (j = 0; j < 2; j++)
      a[i][j] = 1;
```

Mapping

β programs

(a) Code generation for a mapping with out distributed and replicated loops

$$T = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad o = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

```
dist _a = 0 to _np[0] with <a,0>
   dist _b = 0 to _np[1] with <b,0>
<0,0> if (_id[0] == 2 & _id[1] == 1)
<0,0>    for (i = m; i < n; i++)
<0,0>       for (j = 0; j < 2; j++)
<0,0>          a[i][j] = 1;
```

(b) Code generation for mapping with distributed loops

$$T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad o = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
      dist _a = 0 to _np[0] with <a,0>
      rep _b = 0 to _np[1] with <0,b>
<0,0>    for (i = comp_slice_begin(_id[0],m,1,0);
               i < comp_slice_end(_id[0],n,1,0);
               i++)
            dist _c = 0 to _np[1] with <0,c>
<0,0>          for (j = comp_slice_begin(_id[1],0,1,0);
                        j < comp_slice_end(_id[1],2,1,0);
                        j++)
<0,0>             a[i][j] = 1;
```

(c) Code generation for mapping with replicated loops

$$T = \begin{bmatrix} * & 0 \\ 0 & * \end{bmatrix} \quad o = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

```
         rep _a = 0 to _np[0] with <a,0>
         rep _b = 0 to _np[1] with <b,0>
<0,0>    for (i = m; i < n; i++)
            rep _a = 0 to _np[0] with <a,0>
            rep _b = 0 to _np[1] with <b,0>
<0,0>       for (j = 0; j < 2; j++)
               rep _a = 0 to _np[0] with <a,0>
               rep _b = 0 to _np[1] with <b,0>
<0,0>          a[i][j] = 1;
```

sion 1, followed by the outer loop. On the second iteration of the loop in phase 3, we emit a `dist` for dimension 1 and remove it from *DIST*. We don't emit any `rep`'s, and then we emit the loop. In phase 4, we emit the body.

In part (c) we have an example of a loop that is replicated on all processors. After phase 1, REP contains 0 and 1, DIST is empty, and NDIST is empty. Since NDIST is empty, we skip phase 2. In phase 3 for the first iteration of the loop, we emit `rep`'s for the 0 and 1 dimension, we then emit the outer loop unchanged. In the second iteration, we do the same thing. In phase 4, we emit `rep`'s for the 0 and 1 dimension, followed by the body.

5.3.3 Debugging

With respect to debugging, iteration distribution mainly alters the control flow. The control flow is changed by the addition of extra code to ensure that every processor executes it assigned iterations. We discuss control flow first, and then describe data structures afterwards.

For the run and where commands, the *D* function must make it appear that the control flow of the target program is the same as the control flow of the source program. When executing the target program, the debugger must skip the extra statements that are executed in the β program but do not correspond to statements executed in the source program. We can skip statements by repeatedly single stepping until we are past them, which can be done if the compiler-inserted statement cannot cause exceptions.

There are two types of statements that the β program executes that do not correspond to statements in the α program. The first are `if` statements inserted by phase 2 of Algorithm 5.1. These can be easily recognized. The second type are called inactive loops. Inactive loops are assigned iterations to execute that are all below the low bound or all above the high bound of the original loop in the α program.

To explain why inactive loops occur, we use the example in FIGURE 5-8. In this figure, we have expanded the program from part (b) of FIGURE 5-7 assuming a 2x2 processor array. A processor in row *i* executes iteration *i* of the outermost loop. If the low bound of the outer loop has a value of 1, then processors in row 0 do not execute any iterations. In this case, the loop on line 1 is an inactive loop. If the high bound of the outer loop is 0, the processors in row 1 do not execute any iterations. In this case, the loop on line 12 is an inactive loop. An inactive loop is detected by comparing the low and high bounds for the original loop to the low and high bounds for a distributed loop. If the bounds for a loop are completely outside the original bounds, then the loop is inactive. From the example, if the original low bound (`m`) is 1 and the high bound (`n`) is 1, then the loop on line 1 is inactive because its high bound is 0. Being inactive is not a static attribute; the status can change every time a loop is executed.

For data, we exclude modifying loop counters while debugging. Changing a loop counter of a distributed loop would require that we also change the current position in the program. For example, if we set the loop counter to iteration 1, we would also have to change the current position to the loop which executes iteration 1 (use the *T* matrix to compute which statement).

FIGURE 5-8 β program from part (b) of FIGURE 5-7 expanded for a 2x2 processor array

```
1  <0,0>,<0,1> for ( i = comp_slice_begin(_id[0],m,1,0);
2                    i < comp_slice_end(_id[0],n,1,0);
3                 i++)
4  <0,0>          for ( j = comp_slice_begin(_id[1],0,1,0);
5                        j < comp_slice_end(_id[1],2,1,0);
6                        j++)
7  <0,0>             a[i][j] = 1;
8  <0,1>          for ( j = comp_slice_begin(_id[1],0,1,0);
9                        j < comp_slice_end(_id[1],2,1,0);
10                       j++)
11 <0,1>             a[i][j] = 1;
12 <1,0>,<1,1> for ( i = comp_slice_begin(_id[0],m,1,0);
13                   i < comp_slice_end(_id[0],n,1,0);
14                   i++)
15 <1,0>          for ( j = comp_slice_begin(_id[1],0,1,0);
16                       j < comp_slice_end(_id[1],2,1,0);
17                       j++)
18 <1,0>             a[i][j] = 1;
19 <1,1>          for ( j = comp_slice_begin(_id[1],0,1,0);
20                       j < comp_slice_end(_id[1],2,1,0);
21                       j++)
22 <1,1>             a[i][j] = 1;
```

The *D* function for iteration distribution can be found below. The **run** command takes a list of statements in the breakpoint list and sets a breakpoint for every copy of the statement (there is one copy on each processor). It then runs the program. If the program stops at an inserted statement or an inactive loop, then it single steps until it reaches a statement that is not. Single stepping over loops that are inactive may not be possible if the execution of the loop control causes an exception. The debugger must pick the first processor that does not execute an inactive loop to report to the user as the current line. This case is not included in the *D* function. The **where** command simply peels off the processor index off the label when reporting the current location. The **set** command prevents the user from modifying a loop counter and the **examine** command just passes the command through to the lower level debugger.

```
DIterationDistribution(dstack,command,state,program,bpts,name,value)
{
    newbpts = ∅
    if (command == run) {
        foreach label l ∈ bpts
            foreach processor index p
            newbpts = newbpts ∪ (p,l)
        newstate = apply(first(dstatck),rest(dstack),run,state,p,nb,⊥,⊥)
        if we stop at an inserted statement or an inactive loop
            single step until we reach a statement that is neither
        return newstate
    }
    if (command == where) {
        w = apply(first(dstatck),rest(dstack),where,state,⊥,⊥,⊥)
        peel the processor index off w and return just the label
    }
    if (command == examine) {
        return apply(first(dstack),rest(dstack),examine,state,⊥,name,⊥);
    }
    if (command == set) {
        if (name is a loop counter for a distributed loop)
            set is not allowed
        else return apply(first(dstatck),rest(dstatck),set,state,⊥,name,value);
    }
}
```

5.4 Communication

In this section, we describe how the compiler can insert communication into the parallel program and how it affects the debugger. Communication between processors is necessary when one processor must access data that is local to another processor. Because we only study the problem of finding bugs in the user program, communication has little impact on debugging. `send` and `receive` operations are not visible to the user in the original sequential program and thus communication code can be ignored by the debugger. We assume that the compiler inserts communication into the program correctly and that communication never fails.

The only difficulty lies in code generation of the β program; because there is only one thread of control, sends and receives must be inserted into the β program so that a receive is always executed after the corresponding send. This requirement can complicate the structure of some β programs.

The compiler specifies communication by giving the position in the loop nest and the code to perform the communication. Sometimes it is only necessary to communicate at the beginning or end of processing a block of iterations rather than on a per-iteration level. Since the blocking of iterations is not visible in the source code, the compiler cannot name a position to insert the code if the action is only to be performed once per block. For a per-block action, the compiler must give the nesting level of the loop and whether the action is to be performed before or after processing of a slice.

A communication action is performed on every processor that the containing iteration is performed on; if the iteration is replicated across a dimension, so is the communication action.

If a variable reference in a loop body is non-local, then that value must be received from another processor during the course of the computation. The variable reference must be replaced with the name of the temporary in which the value is received. In addition to the communication actions, the compiler also gives a set of variable references and the expressions which replace them.

5.4.1 Code generation

send's and **receive**'s are mapped to the same processors as the rest of the loop iteration. In general, communication statements inherit the processor index label of the surrounding iteration. Per-block communication is treated the same as other communication. It is inserted inside the **dist**: before the loop if it precedes the block and after the loop if it follows the block. In the body of the loop, the appropriate array references are replaced with the substitute references.

5.4.2 Examples

In this section we will give examples of common usages of communication in programs. We will show the user code, the generated code, and discuss how to demonstrate that the program can still be executed sequentially after the communication is inserted.

One use of communication is to transmit values that are produced in one iteration and consumed in a later one. If the iterations are mapped to different address spaces, then the value must be sent from the processor that generates the value to the processor that uses it. As an example, consider the following loop that sums the elements of an array:

```
sum = 0;
for (i = 0; i < n; i++) {
   sum = sum + C[i]
}
```

There is a flow dependence: the value of **sum** used in the right hand side is the value stored in the left hand side in the previous iteration. If the iteration distribution is $T = \begin{bmatrix} 0 & 1 \end{bmatrix}$ and $o = \begin{bmatrix} 0 \end{bmatrix}$, successive iterations are mapped to successive processors and the value of **sum** must be sent from processor to processor. Communication code would be inserted as follows:

```
   rep a = 0 to _np-1 with <a>
<0>   sum = 0;
   dist a = 0 to _np-1 with <a>
<0>   for ( i = comp_slice_begin(_id,0,1,0);
            i < comp_slice_end(_id,n,1,0);
            i++) {
<0>       receiven((-1),sum)
<0>       sum = sum + C[i]
<0>       sendn((1),sum)
      }
```

Each cell receives `sum` from its left neighbor, adds to it, and passes it on to its right neighbor. Recall from CHAPTER 2 that the semantics of `receivon` and `sendn` imply that the first cell should not receive anything and the last cell should not send anything. This communication is used to satisfy a flow dependence. Flow dependences always go forward in virtual time because the use must occur after the generation of the value in the β program.

Another common communication usage is a broadcast operation. The code below is mapped to a 1 dimensional processor array:

```
for (i = 0; i < n; i++)
   c = b[i];
```

The iteration distribution is $T = [*]$ and $o = [0]$ (every processor executes every iteration), and the array `b` is distributed. We explain data distribution in the next section; assume that processor n has array element `b[n]`. Every processor executes every iteration of the outer loop, thus it needs every element of the array `b`. For every iteration of the `i` loop, the processor that has `b[i]` must broadcast its value to all the other processors.

```
   rep a = 0 to _np[0] with <a>
<0>   for (i = 0; i < n; i++) {
         rep a = 0 to _np[0] with <a>
<0>         bcast(b[i],id[0]==i,tmp);
         rep a = 0 to _np[0] with <a>
<0>            c = tmp;
      }
```

In a broadcast, the selected processor sends out its copy of the data and all the other processors must receive it. It is not necessary to make the actual `send` and `receive` statements in a broadcast visible to the user who wrote the α program, so we use the `bcast` function to hide the details of the broadcast operation. The first argument is the data item to be broadcast, the second argument is the expression that is true on the processor that is the source of the broadcast, and the third argument is the destination variable of the broadcast. The reference to `b[i]` in the loop body is replaced with `tmp`. All the sends and receives used to perform one `bcast` operation occur in a single iteration of the loop, so we can be sure that there is a sequential execution of the program—the β program does not deadlock.

In the beginning of Section 5.4, we mentioned that if a receive is executed before the corresponding send in a β program, then a ω program that is deadlock free can deadlock while debugging. This is because debugging can force the ω program to execute operations in the same order as the β program, and if a receive is before its corresponding send, the receive never completes. The extra synchronization that the debugger enforces causes the deadlock.

We use the following example to illustrate how communication can *incorrectly* be inserted into a program to create a negative time dependence.

```
for (i = 0; i < n; i++)
   b[i] = a[i] + a[(i+1) % n]
```

If we were to use the iteration distribution of $T = \boxed{1}$ and $o = \boxed{0}$, then the program after iteration distribution would be:

```
   dist a = 0 to _np-1 with <a>
<0>   for ( i = comp_slice_begin(_id,0,1,0);
            i < comp_slice_end(_id,n-1,1,0);
            i++)
<0>         b[i] = a[i] + a[(i+1) % n]
```

Assume that the data is distributed so that processor i has array element `a[i]` and `b[i]`. According to the iteration distribution, processor i is assigned to execute iteration i. To execute its iteration, the processor already has `a[i]` and `b[i]` local, but must fetch `a[i+1]` from an adjacent processor. We could insert communication into the program as follows:

```
   dist a = 0 to _np-1 with <a>
<0>   for ( i = comp_slice_begin(_id,0,1,0);
            i < comp_slice_end(_id,n,1,0);
            i++) {
<0>         send((-1),a[i+1]);
<0>         receive((1),tmp);
<0>         b[i] = a[i] + tmp;
      }
```

This β program deadlocks because iteration i receives a value from its neighbor to the right. Its right neighbor executes iteration $i+1$, so iteration $i+1$ must execute the **send** statement before iteration i can complete its **receive** statement. When debugging, we must be able to execute the iterations in order; if we try to execute iteration i without executing iteration $i+1$, the program deadlocks. However, when executing the ω without the debugger, the program can complete without deadlock. The loop iterations execute in parallel without any synchronization except whatever is forced by the **send** and **receive** statements, so iteration $i+1$ can execute before iteration i.

The correct way to write the β program for this example is to do the communication in a separate loop before the computation loop. Since iteration i is assigned to processor i,

the computation loop is a left to right pass over the array. The communication loop sends values from processor $i+1$ to processor i, which is a right to left pass over the array. Putting the communication and computation in separate loops resolves the ordering conflict, but might result in a slightly less efficient program.

5.4.3 Debugging

Since the communication is not visible in the source program, the debugging function ignores communication statements that have been inserted; when running, it skips over them by repeatedly single stepping.

```
Dcommunication(dstack,command,state,program,bpts,name,value)
{
    if (command == run) {
        newstate apply(first(dstack),rest(dstack),run,state,program,bpts,⊥,⊥)
        if (stopped at a communication statement)
            single step until we reach a statement
                that is not communication
        return newstate
    }
    if (command == where) {
        return apply(first(dstatck),rest(dstack),where,state,⊥,⊥,⊥)
    }
    if (command == examine) {
        return apply(first(dstack),rest(dstack),examine,state,⊥,name,⊥);
    }
    if (command == set) {
        return apply(first(dstack),rest(dstack),set,state,⊥,name,value);
    }
}
```

5.5 Data distribution

When we distribute iterations of loops to processors, we must also distribute the data that those loop iterations access. Data distributions are specified in the same way loop iteration distributions are specified; there is a linear mapping between the elements of multidimensional data arrays and the multidimensional processor arrays. Data can be replicated by using "*" in the mapping. As before, we limit the type of distribution to the case where there is at most one non-zero element in every row and every column of the distribution matrix. In effect, data arrays are decomposed along one or more dimension and distributed as slices of the original data array. Array references will be represented by a vector of the subscripts for each dimension. For example, reference `a[i][j+i][1]` is:

$$\begin{bmatrix} i \\ j+i \\ 1 \end{bmatrix}$$

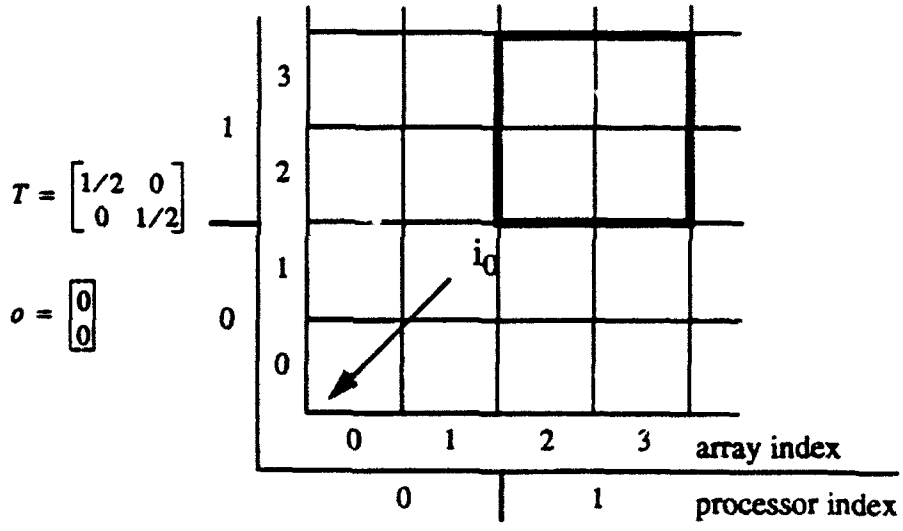
The mapping of array elements to processor indexes is identical to the mapping of iterations to processor indexes. The processor index of an array element can be computed from the array index as follows:

Equation 5.2

$$p = \lfloor T(i) + o \rfloor$$

where T is the transformation matrix and o is the offset. In the example in FIGURE 5-9, an array is divided into 2x2 slices as determined by the T matrix. Processor $\begin{bmatrix}1\\1\end{bmatrix}$ gets the slice containing $\begin{bmatrix}2\\2\end{bmatrix}$, $\begin{bmatrix}2\\3\end{bmatrix}$, $\begin{bmatrix}3\\2\end{bmatrix}$, and $\begin{bmatrix}3\\3\end{bmatrix}$.

FIGURE 5-9 Relationship between global and local array addresses



A *global address* is the set of array indexes of an array element in the sequential program; a *local address* is the set of array indexes used in one processor of the parallel program. To convert the global address to a local address, we subtract a constant offset from all addresses so that the location of the element in the slice with the smallest address, which we shall call i_0, is translated to the origin. In the example of FIGURE 5-9, we would subtract $\begin{bmatrix}2\\2\end{bmatrix}$ from an array reference on processor (1,1) to convert a global address to a local address. For some types of mappings it is necessary to locate i_0 at some point other than the origin. For example, we might want to leave row 0 unused so that we can put a copy of a row there that has been mapped to an adjacent processor. We call this offset s.

In general, the relationship between global and local addresses on a processor p is as follows:

Equation 5.3

$$l = g - i_0(p) + s$$

where l is the local address, g is the global address, and $i_0(p)$ is the address of the array element with the smallest lexicographic value that is on processor p.

i_0 can be computed from the mapping matrix T, offset o, and Equation 5.2. Given a p, we want to find the minium value of i such that $p = \lfloor T(i) + o \rfloor$. The value of i_0 is found by computing $T'(p-o)$ where T' is found by inverting all the non-zero elements of T, replacing all *'s with 0, and then taking the transpose. For example:

$$T = \begin{bmatrix} 0 & 0 & * \\ 0 & 0 & 0 \\ 1/2 & 0 & 0 \end{bmatrix} \qquad T' = \begin{bmatrix} 0 & 0 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Note that we cannot simply solve Equation 5.2 for i because neither the floor function nor T are necessarily invertible. If we substitute the value for i_0 in Equation 5.2 we obtain:

Equation 5.4

$$l = g - T'(p-o) + s$$

5.5.1 Code generation

When generating the β program, the indexes for references to distributed variables must be rewritten to reflect their new location in memory. For the body of the loop nest, we will assume that all array references to distributed variables will reference data elements that are local to the processor. If it were necessary to access a non-local element, then this would have been replaced with a reference to a local variable during the stage that adds communication to a program.

5.5.1.1 Example

If the 3 dimensional array B were mapped to a 2 dimensional processor array as follows:

$$T = \begin{bmatrix} 0 & 1/3 & 0 \\ 1/2 & 0 & 0 \end{bmatrix} \qquad o = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad s = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \text{ then}$$

$$T' = \begin{bmatrix} 0 & 2 \\ 3 & 0 \\ 0 & 0 \end{bmatrix} \qquad p = \begin{bmatrix} \_id[0] \\ \_id[1] \end{bmatrix}$$

We can use Equation 5.4 to convert the global address to a local address. If we substitute the above values into Equation 5.4, we obtain:

$$l = g - T(p - o) + s$$

$$= g - \begin{bmatrix} 0 & 2 \\ 3 & 0 \\ 0 & 0 \end{bmatrix} \left(\begin{bmatrix} \_id[0] \\ \_id[1] \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$= g - \begin{bmatrix} 2 \times \_id[1] - 2 \\ 3 \times \_id[0] \\ 0 \end{bmatrix}$$

This is used for any reference to the array `B`. The array subscript `B[i][j+k][2]` would be remapped to:

$$l = g - \begin{bmatrix} 2 \times \_id[1] \\ 3 \times \_id[0] - 1 \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} i \\ j+k \\ 2 \end{bmatrix} - \begin{bmatrix} 2 \times \_id[1] - 2 \\ 3 \times \_id[0] \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} i - 2 \times \_id[1] + 2 \\ j + k - 3 \times \_id[0] \\ 2 \end{bmatrix}$$

which in the program would appear as:

```
B[i-2*_id[1]+2][j+k-3*_id[0]][2]
```

The more complicated addressing needed to reference distributed variables is not necessarily inefficient. Note that the variable `_id` is loop invariant so the entire offset is loop invariant and need not be computed inside the loop. Furthermore, every reference to the same variable will have the same offset, so the offset must be computed at most once for every distributed variable before the loop begins.

5.5.2 Debugging

Distributing data does not directly change the structure of the program, thus the flow control is unchanged. If the user wants to examine or modify data, the debugger must determine which processor has the data and the local address of the data on that processor.

The debugging function for the data distribution transformation is as follows:

```
DDataDistribution(dstack,command,state,program,bpts,name,value)
{
    if (command == run) {
        return apply(first(dstack),rest(dstack),run,state,program,bpts,⊥,⊥)
    }
    if (command == where) {
        return apply(first(dstack),rest(dstack),where,state,⊥,⊥,⊥)
    }
    if (command == examine) {
        if (name is distributed)
            return apply(first(dstack),rest(dstack),
                        examine,state,⊥,global2local(name),⊥);
        else { /* copy to look at is on same processor as the current statement */
            w = apply(first(dstack),rest(dstack),where,state,⊥,⊥,⊥)
            i = processor index of current statement w
            return apply(first(dstack),rest(dstack),examine,state,⊥,(name,i),⊥);
        }
    }
    if (command == set) {
        if (name is distributed)
            return apply(first(dstack),rest(dstack),
                        set,state,⊥,global2local(name),value);
        else {
            foreach processor i { /* change all copies */
                state = apply(first(d),rest(d),set,s,⊥,(name,i),v);
                return state
            }
    }
}
```

The function global2local uses Equation 5.2 to compute the processor number and Equation 5.4 to convert the global address to a local address.

5.5.2.1 Example

A three dimensional array A is distributed as follows on a 2 dimensional processor array:

$$T = \begin{bmatrix} 0 & * & 0 \\ 0 & 0 & 1/2 \end{bmatrix} \qquad o = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad s = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$ then:

$$T = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 2 \end{bmatrix}$$

The users wants to inspect the variable A[1][2][11]. To compute the processor index:

$$p = \lfloor T(g) + o \rfloor$$

$$= \left\lfloor \begin{bmatrix} 0 & * & 0 \\ 0 & 0 & 1/2 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 11 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\rfloor$$

$$= \begin{bmatrix} * \\ 6 \end{bmatrix}$$

To compute the local address:

$$l = g - T'(p - o) + s$$

$$l = \begin{bmatrix} 1 \\ 2 \\ 11 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 2 \end{bmatrix} \left(\begin{bmatrix} * \\ 6 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$l = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

The value can be found on any processor in column 6 with a local address of `A[1][2][1]`.

5.6 Cyclic iteration distributions

A linear mapping model, as described in the previous sections, does not always perform well when the work is not uniformly distributed across iterations. For example, the work performed by one iteration of the outer loop of the nest in FIGURE 5-2 increases with every iteration. If outer loop iterations are distributed evenly across a one dimensional processor array, then the last processor does much more work than the first processor; the load balancing is poor. Any linear model that parallelizes the outer loop must divide the iterations equally among processors.

One mapping model that is used to better distribute the load for such a case is called a cyclic distribution. In this model, a processor will execute every n iterations, where n is usually the number of processors. On an n processor array, processor p would execute iterations p, $p+n$, $p+2n$, ... It is also possible to cyclically distribute blocks of iterations in the same way. If the number of iterations is sufficiently larger than the number of processors, then each processor will have a nearly equal amount of work for triangu-

lar problems. Cyclic distributions have disadvantages, however. If iteration n produces a value that is used by iteration $n+1$, then the value must be sent from one processor to another for a block size of 1. If a cyclic distribution were not used, both iterations could be part of the same block and assigned to the same processor, reducing the frequency of communication.

Cyclic distribution of loop iterations usually requires that we distribute the data arrays in the same way. We will describe distributing iterations first and data second.

5.6.1 Iteration distribution

We can extend the description of the mapping of iterations to processors to include cyclic by rewriting Equation 5.1 as:

Equation 5.5

$$p = \lfloor T(i) + o \rfloor \bmod n$$

where mod is an elementwise modulo operation and n is a vector that contains the size of the array in every dimension.

The algorithm for generating code and D functions as presented previously will no longer work because the modulo function makes the mapping non-linear. Instead of adapting the code generation and D functions to include cyclic distributions, we will do a transformation on the source program before generating the β program, and then use the standard code generation algorithm. The transformation changes the shape of the iteration space in a manner that makes it possible to use a linear mapping to do the cyclic distribution.

The user gives a T and o assuming Equation 5.5 will be used for the mapping. A transformation is applied to the loop nest to generate another loop nest, $\hat{T}$, ∂, and a D function. The code generation algorithm (Algorithm 5.1) is applied to the new loop nest, using $\hat{T}$ and ∂ to specify the mapping.

5.6.1.1 Code generation

We can extend the mapping model to include the cyclic distribution by applying a loop transformation called strip mining[Padua 86] to the original sequential program. An example in FIGURE 5-10 is used to illustrate this. The left hand side of the figure is a loop and its triangular iteration space. The `i` dimension is the vertical axis and the `j` dimension is the axis coming out of the page. If we apply strip mining to the outer loop, then that loop will be replaced with two loops. Inside the body of the loop, the value of the original loop counter can be computed by summing the two new loop counters. The new loop and iteration space is on the right hand side of the figure. The `i` dimension is replaced with the `tmp` and `newi` dimensions, which are horizontal and vertical, respectively.

After strip mining is applied, a linear mapping can be used to achieve the desired assignment of iterations to processors. In FIGURE 5-11, a mapping of $T = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$ and $o = \begin{bmatrix} 0 \end{bmatrix}$ could be used to achieve a cyclic mapping of iterations to processors.

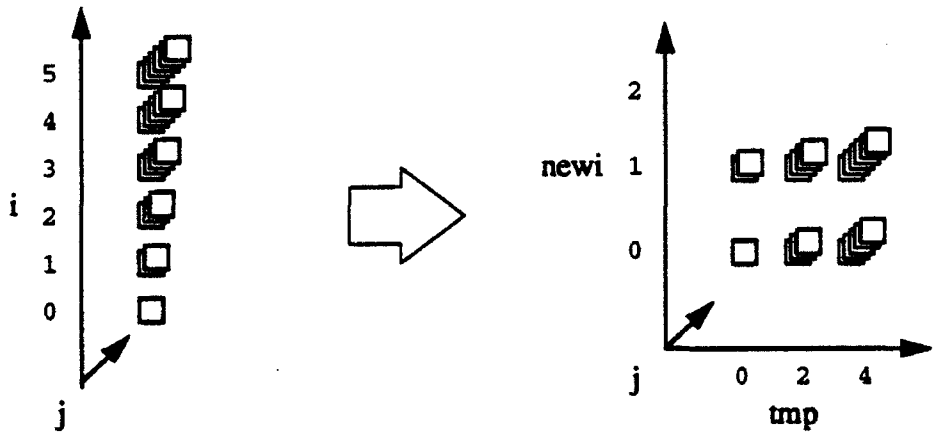
FIGURE 5-10 Triangular iteration space that is changed by strip mining

Original loop and iteration space

```
for (i = 0; i < 6; i++)
   for (j = 0; j < i; j++) ...
```

Loop and iteration space after strip mining

```
for (tmp = 0; tmp < 6; tmp += 2)
   for (newi = 0; newi < 2; newi++)
      for (j = 0; j < i; j++)
         (i = tmp + newi)...
```



The general procedure for the cyclic distribution is as follows. The input is a loop nest where the *m*th loop in the nest is of the form:

```
for (k = l; k < h; k += s)
```

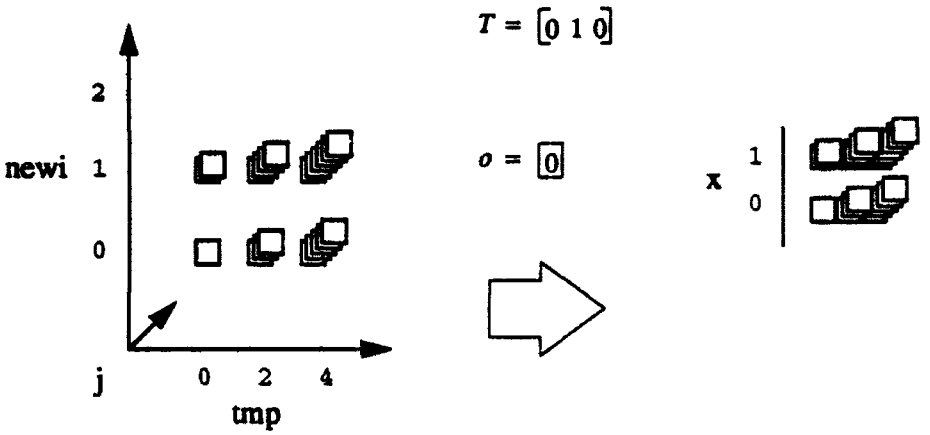
T and *o* are the parameters to the mapping. The *m*th column of *T* is the column that describes the mapping of loop *m*. Since this loop is distributed, it has one non-zero element *c* at position *r*. A new nest will be output which is the same except loop *m* is replaced with the code below. The variable `k`, `l`, `h`, and `s` refer to the loop above.

```
newl = ((((c*l+o[m])/_np[r])*_np[r])-o[m])/c;
for (_t = newl; t < h; t += _np[r]/c)
   for ( k = max(0,l-t);
         k < _np[r]/c - max(0,t+_np[r]/c - h);
         k += s)
```

The original loop counter can be reconstructed by adding `_t` and `k`. Uses of the loop counter in the rest of the loop must be replaced with `_t + k`.

The complicated loop control arises from the requirement that it be possible to use a linear mapping for the new loop nest. For a linear mapping, the iterations of a loop may not

FIGURE 5-11 Mapping of the iteration space after strip mining



"wrap around" from the end of the processor array to the beginning; the assignment of iterations to processors must be strictly monotonic; from low indexes to high indexes or high indexes to low indexes. For the rest of this paragraph we assume that mappings only go from low indexes to high indexes. To ensure that wrap around does not occur, we must make sure that the iteration of the inner loop where `k = 0` is always assigned to the first processor in that dimension, which in some cases requires padding by adding extra iterations on to the beginning of the original loop. The variable `newl` is the new low bound that has been adjusted for padding. However, we do not want to really execute these extra iterations, so we use the `max` condition in the low bound computation in the inner loop to skip over those iterations. The `max` condition on the high bound of the inner loop ensures that we do not execute any extra iterations after the original high bound.

The new T matrix is the same as the old one except that an extra column of zero's is inserted into the mth position; the mth column (the new one) is the mapping for the new outer loop (it isn't distributed) and column $m+1$ determines the mapping of the inner loop. The padding of iterations makes an offset unnecessary, so the o vector is the same except that a 0 is used for the processor dimension that the loop is distributed along.

5.6.1.2 Debugging

When single stepping, the debugger must skip over the outer loop. Modifying a loop counter is possible for this transformation, but since the iteration distribution which follows does not permit it, there is no reason to do it for this transformation. If the user examines the loop counter, the debugger simply returns the sum of the two new loop counters.

5.6.2 Data distribution

Now we will describe how to do cyclic data distributions. It is very similar to cyclic iteration distributions. We can extend the description of the mapping of data elements to processors by rewriting Equation 5.2 as:

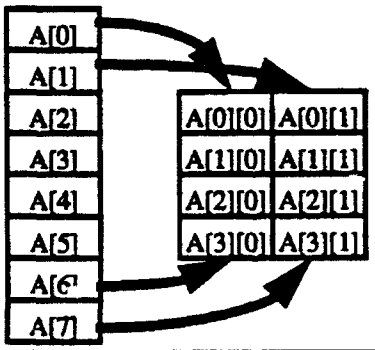
Equation 5.6

$$p = \lfloor T(i) + o \rfloor \bmod n$$

where mod is an elementwise modulo operation and n is a vector that contains the size of the array is every dimension.

In the same way that we do a cyclic distribution of loop iterations by turning a n deep loop nest into a $n+1$ deep loop nest, we will distribute data by turning a n dimensional data array into a $n+1$ dimensional data array before we do the distribution. An example is shown in FIGURE 5-12. The one dimensional array A with 8 elements has been changed into a 2 dimensional array that is 4 rows by 2 columns. Some of the mappings from source elements to target elements are denoted by arrows from one object to another.

FIGURE 5-12 Converting a 1 dimensional array into a 2 dimensional array in preparation for a cyclic distribution.



5.6.2.1 Code generation

All references to a distributed data array must be rewritten to reflect the new structure. The general procedure is as follows. The input is an array reference where the mth index in the reference is of the form: `[e]`. T and o are the parameters to the mapping. The mth column of T describes the mapping of index i in the reference. Since this dimension will be distributed, it has one non-zero element c at position r. A new reference will be output which is the same except subscript m is replaced with the two subscripts below:

```
[(c*e+o[m])/_np[r]][e-(c*e+o[m])/_np[r]]
```

The new T matrix is the same as the old one except that an extra column of zero's is inserted into the mth position; the mth column (the new one) is the mapping for the new first index and column $m+1$ determines the mapping of the new second index. The

padding of data elements makes an offset unnecessary so the o vector is the same except that a 0 is used for the processor dimension that the array dimension is distributed along. The s vector, which is used to compute the local address, is unchanged.

The address computation appears to be much more complicated than the original reference, but this will not necessarily result in less efficient code. Array references in loops like these are usually linear functions of loop counters, which can be generated efficiently. The new indexes are linear functions of the old index so the new indexes will still be linear functions of loop counters if the original index was one. In this case, it might be more expensive to initiate the loop, but the cost per iteration to compute array references need not increase.

5.6.2.2 Debugging

The D function for this transformation is straightforward; the control flow is not altered, only array references are changed. When the user uses the debugger to examine or modify a variable, it must map the source reference into a target reference using the procedure described in the previous section.

5.7 Summary

This chapter introduces a notation for describing the translation from sequential to parallel. There are three parts, the mapping of iterations to processors, the inter-processor communication, and the mapping of data to processors. The description contains the information that the debugger needs to generate the D function for the sequential transformation and the β program. The notation is general enough to do the block and cyclic distributions of data and iterations that most parallelizing compilers employ.

CHAPTER 6

A debugger for a compiler with block and cyclic distributions

The previous two chapters describe the components for the compiler and debugger for the thread splitting and distribution transformations. In this chapter, we integrate those components and demonstrate how they work together for block and cyclic distributions.

We first introduce a matrix multiply program in Section 6.1, which is used as an example throughout the chapter. In Section 6.2, we use matrix multiply to show a complete translation from α to ω for block and cyclic distributions. In Section 6.3, we describe the information that is passed from the compiler to the debugger. We then show how that information is used by describing some of the basic debugging operations in Section 6.4. We conclude by examining some performance issues related to debugging in Section 6.5.

6.1 Matrix multiply

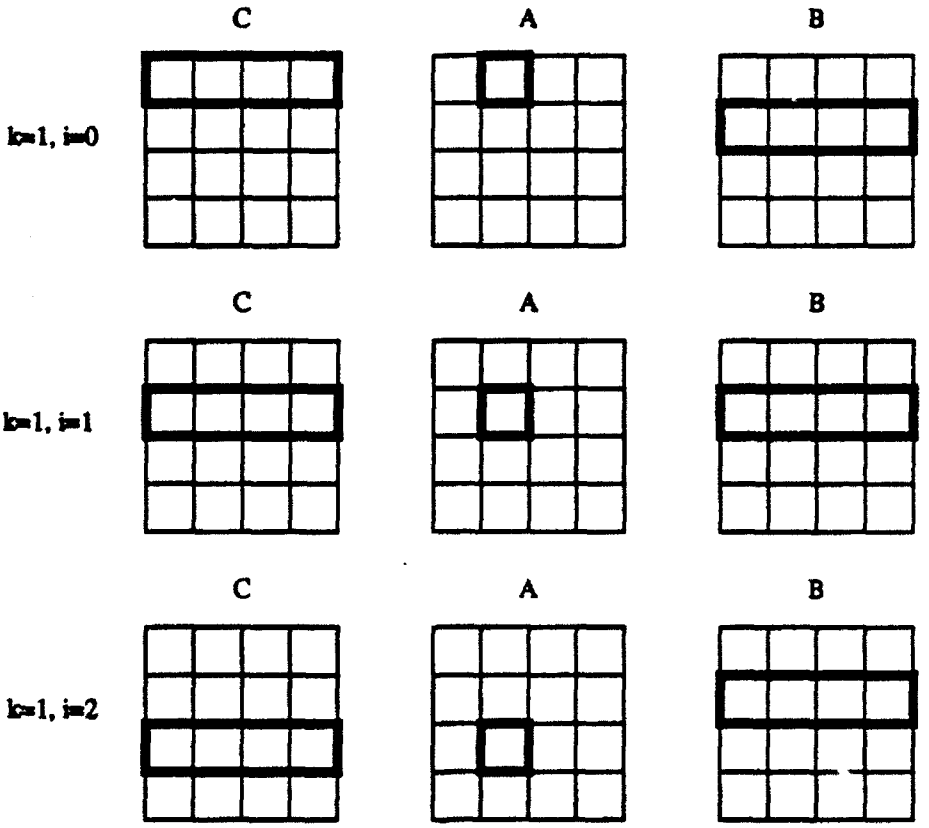
The code for matrix multiply is depicted below. To simplify the code, we don't include any initialization, we assume that the `A`, `B`, and `C` arrays are properly initialized elsewhere. The program is a triple nested loop, each loop will be called by the name of the loop counter: `k`, `i`, and `j`. The `i` and `j` loops have no dependences, so they can be executed in parallel, but the `k` loop has a dependence.

```
11:   for (k = 0;k < d;k++)
12:      for (i = 0;i < d;i++)
13:         for (j = 0;j < d;j++)
14:            C[i][j] += A[i][k]*B[k][j];
```

A compiler or user chooses the distribution of computation and data to minimize communication while distributing work. The distribution determines what communication is

necessary. There are many ways to choose to parallelize matrix multiply; we only describe one way. The parallelization decision is driven by the way the loops access data. FIGURE 6-1 illustrates how matrix multiply accesses its data for the k=1 iteration of the outer loop and the first few iterations of the i loop. Iteration *m* of the i loop only accesses row *m* of the arrays C and A. If we distribute the i loop across processors and divide C and A by rows so that row *m* is on the same processor that executes iteration *m* of the i loop, then we never need to move the C or A arrays between processors. Iteration n of the k loop accesses row *n* of the B array, so every time the program begins the i loop every processor needs a copy of the same row (*n*) of the B array. If the B array is divided by rows, then the processor that has row *n* must broadcast it before beginning the i loop.

FIGURE 6-1 Data accesses of matrix multiply



We start by describing the parallelization that uses a block distribution. The A, B, and C arrays are distributed by row. Each processor gets 2 rows of each array (processor *m* gets row $2m$ and $2m + 1$). All processors execute every iteration of the k loop. For each iteration of the k loop, processor *m* executes iterations $2m$ and $2m + 1$ of the i loop.

Since the iterations and rows of A and C are distributed the same way, no communication is needed for these arrays. Before a processor begins executing iteration m of the i loop, it must fetch row n of the B array, where n is the current value of the loop counter k. Each processor executes all the iterations of the j loop.

To motivate the steps in the translation, we first we show the final parallel code; in the next section we go through the steps of generating it.

```
for (k = 0;k < d;k++) {
   for (t = 0; t < d; t++)
      bcast(_id==k/2,b[k-2*_id][t],tmp[t]);
   for (i = 2*_id; i < 2*_id+1;i++)
         for (j = 0;j < n;j++)
            C[i-2*_id][j] += A[i-2*_id][k]*tmp[j];
}
```

The bcast primitive implements a broadcast for a single element. The first argument is true if the processors should broadcast the value, or false if the processor is a recipient of the broadcast. The second argument is the value to be broadcast. This value is ignored on the processors that are receiving the broadcast value. The third argument is the location that should receive the broadcast value. It is ignored on the processor that sends the broadcast value. The variable _id contains the processor index.

The cyclic distribution is similar to the block distribution. The A, B, and C arrays are again distributed by row. The difference is in the rows and iterations that are assigned to each processor. Each processor gets 2 rows of each array (processor m gets row m and $m+p$, where p is the number of processors). All processors execute every iteration of the k loop. For each iteration of the k loop, processor m executes iteration m and $m+p$ of the i loop. Since the iterations and rows of A and C are distributed the same way, no communication is needed for these arrays. Before a processor begins executing iteration m of the i loop, it must fetch row n of the B array, where n is the current value of the loop counter k. Each processor executes all the iterations of the j loop.

The final parallel code is as follows:

```
for (k = 0;k < d;k++) {
   for ( t = 0; t < d; t++)
      bcast(_id==k%_np,b[k%_np-_id][t],tmp[t]);
   for (t = 0; t < d; t += _np)
      for (i = 0; i < _np; i++)
         for (j = 0;j < n;j++)
            C[t/_np][i][j] += A[t/_np][i][k]*tmp[j]
}
```

The variable _np contains the number of processors. The difference between the two programs is in the way local addresses are computed and the bounds for the i loop.

6.2 Compilation

Now we show the compilation process for matrix multiply using block and cyclic distributions. The steps are iteration distribution, inserting communication, data distribution, and thread splitting. The resulting program is compiled by a single cell compiler which generates the executable.

6.2.1 Block distribution

We start with the block distribution. The distribution is specified by the mapping of iterations to processors, the mapping of data to processors, and the communication. We start with the mapping of iterations to processors. We want to give each processor 2 iterations of the `i` loop, so the iteration distribution is: $T = \boxed{0\ 2\ 0}, o = \boxed{0}$. When we apply this distribution to the program by using Algorithm 5.1, we obtain the following program:

```
      rep a = 0 to _np-1 with <a>
<0>   for (k = 0;k < d;k++)
         dist b = 0 to _np-1 with <b>
<0>      for ( i = comp_slice_begin(_id,0,2,0);
               i <= comp_slice_end(_id,d,2,0);
               i++)
<0>         for (j = 0;j < d;j++)
<0>            C[i][j] += A[i][k]*B[k][j];
```

In the next phase, the compiler inserts communication. As was described in the previous section, for iteration n of the `k` loop, the processor that has row n of the `B` array must broadcast that row to all processors before beginning the i loop. For this reason, the compiler inserts a broadcast before loop i. The references to the array `B`, must be to the local copy received, so the compiler also replaces references to `B[k][j]` in the loop with `tmp[j]`. After communication is inserted, we have the following program:

```
      rep a = 0 to _np-1 with <a>
<0>   for (k = 0;k < d;k++) {
         rep a = 0 to _np-1 with <a>
<0>      for (t = 0; t < d; t++)
            rep a = 0 to _np-1 with <a>
<0>         bcast(2*_id==k,B[k][t],tmp[t]);
         dist b = 0 to _np-1 with <b>
<0>         for ( i = comp_slice_begin(_id,0,2,0);
                  i <= comp_slice_end(_id,d,2,0);
                  i++)
<0>            for (j = 0;j < d;j++)
<0>               C[i][j] += A[i][k]*tmp[j];
      }
```

Next, the compiler converts global addresses to local addresses. Each of the arrays are distributed so that each processor gets 2 rows. The mapping is specified by:

$$\text{A, B, C: } T = \begin{bmatrix} 2 & 0 \end{bmatrix}, o = \begin{bmatrix} 0 \end{bmatrix}, s = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The compiler uses the above specification and Equation 5.4 to compute the mapping between global addresses and local addresses as follows. Arrays A, B, and C all have the same mapping, so we only need to show the global to local mapping once.

$$T = \begin{bmatrix} 2 & 0 \end{bmatrix} \qquad o = \begin{bmatrix} 0 \end{bmatrix} \qquad s = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad p = \begin{bmatrix} \_id \end{bmatrix}$$

$$T = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

$$l = g - T(p - o) + s$$

$$= g - \begin{bmatrix} 2 \\ 0 \end{bmatrix} \left(\begin{bmatrix} \_id \end{bmatrix} - \begin{bmatrix} 0 \end{bmatrix} \right) + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$= g - \begin{bmatrix} 2 \times \_id \\ 0 \end{bmatrix}$$

In the code we must subtract `2*_id` from the first index of a global address. This gives us the following program:

```
1         rep a = 0 to _np-1 with <a>
2   <0>   for (k = 0;k < d;k++) {
3            rep a = 0 to _np-1 with <a>
4   <0>      for (t = 0; t < d; t++)
5               rep a = 0 to _np-1 with <a>
6   <0>         bcast(2*_id==k,B[k-2*_id][t],tmp[t]);
7            dist b = 0 to _np-1 with <b>
8   <0>         for ( i = comp_slice_begin(_id,0,2,0);
9                         i <= comp_slice_end(_id,d,2,0);
10                        i++)
11  <0>            for ( j = 0;j < d;j++)
12  <0>               C[i-2*_id][j] +=
13                       A[i-2*_id][k]*tmp[j];
14        }
```

In the next step, the compiler applies thread splitting, which removes the PIL's and the `rep` and `dist` constructs to yield:

```
1
2     for (k = 0;k < d;k++) {
3
4        for (t = 0; t < d; t++)
5
6           bcast(2*_id==k,B[k-2*_id][t],tmp[t]);
7
8        for ( i = comp_slice_begin(_id,0,2,0),ni=0;
9              i <= comp_slice_end(_id,d,2,0);
10             i++,ni++)
11             for (j = 0;j < d;j++)
12                C[i-2*_id][j] +=
13                   + A[i-2*_id][k]*tmp[j];
14    }
```

Recall that the thread splitting debugger we presented in the previous chapter assumes that loops are normalized (start at 0 and count by 1). In this example, instead of normalizing the `i` loop, we introduce a new variable called `ni` (normalized i) that starts at 0 and counts by 1 in the `i` loop. Whenever we need the loop count for the `i` loop, we reference this variable instead of the loop counter `i`.

This program is then compiled by a single processor compiler that generates an executable which can be loaded into every processor of the parallel machine.

6.2.2 Cyclic distribution

We want to do a cyclic distribution for the `i` loop and the `A` and `C` arrays. The compiler must strip mine the loop and the arrays before applying the basic distribution. After strip mining, the code is as listed below. We have simplified array index and loop index expressions when possible.

```
for (k = 0;k < d;k++)
   for (_t = 0; t < d; t += _np)
      for (i = 0;i < _np;i++)
         for (j = 0;j < d;j++)
            C[_t/_np][i][j] +=
               A[_t/_np][i][k]*B[k][j];
```

After strip mining, the distribution for the loops are: $T = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}, o = \begin{bmatrix} 0 \end{bmatrix}$. The distribution for the `A` and `C` arrays are identical: $T = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}, o = \begin{bmatrix} 0 \end{bmatrix}, s = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. The distribution for the `B` array is the same as in the previous example: $T = \begin{bmatrix} 2 & 0 \end{bmatrix}$, $o = \begin{bmatrix} 0 \end{bmatrix}, s = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

Next, the compiler applies Algorithm 5.1, to obtain the following program:

```
      rep a = 0 to _np-1 with <a>
<0>   for (k = 0;k < d;k++)
         rep a = 0 to _np-1 with <a>
<0>      for (_t = 0; t < d; t += _np)
            dist b = 0 to _np-1 with <b>
<0>            for ( i = comp_slice_begin(_id,0,1,0);
                       i <= comp_slice_end(_id,_np,1,0);
                       i++)
<0>               for (j = 0;j < d;j++)
<0>                  C[_t/_np][i][j] +=
                        A[_t/_np][i][k]*B[k][j];
```

In the next phase, the compiler inserts communication. The broadcast occurs before the `t` loop executes. In the source program, the compiler decides to insert the broadcast after the `k` loop. The references to the array `B`, must be to the local copy received, so the compiler also replaces references to `b[k][j]` in the loop with `tmp[j]`. After communication is inserted, we have the following program:

```
      rep a = 0 to _np-1 with <a>
<0>   for (k = 0;k < d;k++) {
         rep a = 0 to _np-1 with <a>
<0>      for (t = 0; t < d; t++)
            rep a = 0 to _np-1 with <a>
<0>         bcast(_id%_np==k,B[k][t],tmp[t]);
         rep a = 0 to _np-1 with <a>
<0>      for (_t = 0; t < d; t += _np)
            dist b = 0 to _np-1 with <b>
<0>            for ( i = comp_slice_begin(_id,0,1,0);
                       i <= comp_slice_end(_id,_np,1,0);
                       i++)
<0>               for (j = 0;j < d;j++)
<0>                  C[_t/_np][i][j] +=
                        A[_t/_np][i][k]*B[k][j];
      }
```

Next, the compiler converts global addresses to local addresses. Array `B` has the same mapping as before, while `A` and `C` have a different mapping, so we only need to show the global to local mapping once.

$$T = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \quad o = \begin{bmatrix} 0 \end{bmatrix} \qquad s = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad p = \begin{bmatrix} \_id \end{bmatrix}$$

$$T' = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$l = g - T'(p - o) + s$$

$$= g - \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} (\begin{bmatrix} \_id \end{bmatrix} - \begin{bmatrix} 0 \end{bmatrix}) + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$= g - \begin{bmatrix} 0 \\ \_id \\ 0 \end{bmatrix}$$

In the code we must subtract `_id` from the second index of a global address This gives us the following program:

```
1         rep a = 0 to _np-1 with <a>
2   <0>   for (k = 0;k < d;k++) {
3            rep a = 0 to _np-1 with <a>
4   <0>      for (t = 0; t < d; t++)
5               rep a = 0 to _np-1 with <a>
6   <0>         bcast(_id%_np==k,B[k][t-2*_id],tmp[t]);
7            rep a = 0 to _np-1 with <a>
8   <0>      for (_t = 0; t < d; t += _np)
9               dist b = 0 to _np-1 with <b>
10  <0>            for ( i = comp_slice_begin(_id,0,1,0);
11                       i <= comp_slice_end(_id,_np,1,0);
12                       i++)
13  <0>                for (j = 0;j < d;j++)
14  <0>                   C[_t/_np][i-_id][j] +=
15                           A[_t/_np][i-_id][k]*tmp[j];
16        )
```

In the next step, the compiler applies thread splitting, which removes the PIL's and the `rep` and `dist` constructs to yield:

```
1
2   for (k = 0;k < d;k++) {
3
4     for (t = 0; t < d; t++)
5
6        bcast(_id%_np==k,B[k][t-2*_id],tmp[t]);
7
8     for (_t = 0; t < d; t += _np)
9
10       for ( i = comp_slice_begin(_id,0,1,0),ni=0;
11             i <= comp_slice_end(_id,_np,1,0);
12             i++,ni++)
13          for (j = 0;j < d;j++)
14             C[_t/_np][i-_id][j] +=
15                A[_t/_np][i-_id][k]*tmp[j];
16 }
```

This program is then compiled by a single processor compiler that generates an executable which can be loaded into every processor of the parallel machine. Instead of normalizing the `i` loop, we introduce a new variable, `ni`, that has the normalized count.

6.3 Compiler/debugger interface

In this section, we describe the information that must be passed from the compiler to the debugger. We first list the information necessary for distribution in Section 6.3.1 and the information for thread splitting in Section 6.3.2. We use the matrix multiply examples for both.

6.3.1 Distribution

Examples of the information that the compiler must pass to the debugger for the distribution phase of compilation can be found in FIGURE 6-2. The left hand side is the information for the cyclic distribution and the right hand side is for the block distribution. The rest of the section explains what information is needed for each part of the distribution transformation.

If the compiler uses a cyclic distribution for iterations or data, it strip mines loops or arrays first. The information that the *D* function needs is the position of the loop in the program, and the name of the loop counters for the inner and outer loops. If any variables have been strip mined, the compiler passes the debugger the name of the variable and the dimension that has been split.

For iteration distribution, the compiler must pass the debugger the line number of each distributed loop, the name of the loop counter, the expression for the low bound, and the expression for the high bound. It must also pass the line numbers of any statements that it inserted. In this example, neither examples have inserted lines.

The *D* function for communication single steps over code that has been inserted for communication; the compiler passes these line numbers to the debugger.

FIGURE 6-2 Information passed from the compiler to the debugger for distribution

Cyclic distribution

loop strip mine

loop postion	inner loop	outer loop
8	i	_t

data strip mine

variable name	dimension
A	0

iteration distribution

loop position	counter	low bound	high bound
10	i	0	d

inserted communication

line number
4

data distribution

variable name	T	o	s
A	[0 1 0]	[0]	[0 0 0]
B	[2 0]	[0]	[0 0]
C	[0 1 0]	[0]	[0 0 0]

Block distribution

iteration distribution

loop position	counter	low bound	high bound
8	i	0	d

inserted communication

line number
4

data distribution

variable name	T	o	s
A	[2 0]	[0]	[0 0]
B	[2 0]	[0]	[0 0]
C	[2 0]	[0]	[0 0]

The *D* function for data distribution changes global names to local names for all user commands to examine and modify variables. If the data is distributed, it uses Equation 5.2 to compute the processor index and Equation 5.4 to convert the global address to a local address. For non-distributed variables, it examines the copy of the variable on the processor at the current location. If the user modifies a variable, it modifies all copies. The compiler must pass to the debugger the names of distributed variables and the mapping.

6.3.2 Thread splitting

Example of the information that the compiler must pass to the debugger for the thread splitting phase can be found in FIGURE 6-3. All of the information that the compiler must pass the debugger about thread splitting is concerned with computing the virtual time and the set of early operations.

In the table, there is one line of information for every statement of the program. The virtual time template has been described previously. Read and write are the set of variables that are read or written by statement of the program. The line also contains the `rep` and `dist` statements contained in each statement of the program.

By plugging in the values for the `dist` variable and the loop counters, the debugger can compute the virtual time of a statement. The value of loop variables is taken from the program state, and the value of the `dist` variables for a processor index can be computed from the `rep` and `dist` statements and PIL's in a program. The virtual time template can also be used to compute the set of early operations for a state of the program. With the set of early operations, together with the read and write information for every statement, the debugger can compute if a program has early reads or early writes.

6.4 Debugger

This section explains how the *D* functions for distribution and thread splitting work together to form an entire debugger. We start by giving an example of how virtual time is computed in Section 6.4.1. This is followed by a description of how the debugger commands are implemented assuming a block or cyclic distribution in Section 6.4.2. We then explain how the basic mechanisms for the thread splitting debugger, roll forward and consistent, breakpoints are affected by using a block or cyclic distribution in Section 6.4.3 and Section 6.4.4.

6.4.1 Computing virtual time

Computing the virtual time of a statement is central to the ability to determine the current location and decide if user commands are disallowed. To compute the virtual time of a statement, we need two things: the position in the program, which determines the virtual time template, and the value of the parameters in the template. The parameters are the value of `dist` variables and the value of loop counters. The value of `dist` variables can be computed statically for each statement from the processor index. The value of the counters are determined at run-time.

FIGURE 6-3 Information passed from the compiler to the debugger for thread splitting

Block distribution

line number	virtual time template	reads	writes	PIL	rep and dist statements
1					`rep a = 0 to _np[0]-1 with <a>`
2	(2,k,2,0,0,0,0,0,0)	k,d	k	<0>	
3					`rep a = 0 to _np[0]-1 with <a>`
4	(2,k,4,t,4,0,0,0,0)	t,d	t	<0>	
5					`rep a = 0 to _np[0]-1 with <a>`
6	(2,k,4,t,6,0,0,0,0)	k,t,tmp,B	tmp,B	<0>	
7					`dist b = 0 to _np-1 with <b>{`
8	(2,k,7,b,8,ni,8,0,0)	i,d	i	<0>	
9					
10					
11	(2,k,7,b,8,ni,11,j,11)	j,d	j	<0>	
12	(2,k,7,b,8,ni,11,j,12)	i,j,k,A,tmp,C	C	<0>	
13					}
14					

Cyclic distribution

line number	virtual time template	reads	writes	PIL	rep and dist statements
1					`rep a = 0 to _np[0]-1 with <a>`
2	(2,k,2,0,0,0,0,0,0,0,0)	k,d	k	<0>	
3					`rep a = 0 to _np[0]-1 with <a>`
4	(2,k,4,t,4,0,0,0,0,0,0)	t,d	t	<0>	
5					`rep a = 0 to _np[0]-1 with <a>`
6	(2,k,4,t,6,0,0,0,0,0,0)	k,t,tmp,B	tmp,B	<0>	
7					`rep a = 0 to _np[0]-1 with <a>`
8	(2,k,8,_t,8,0,0,0,0,0,0)	_t,,d	_t	<0>	
9					`dist b = 0 to _np-1 with <b>{`
10	(2,k,8,_t,9,b,10,ni,10,0)	i,d	i	<0>	
11					
12					
13	(2,k,8,_t,9,b,10,ni,13,j,13)	j,d	j	<0>	
14	(2,k,8,_t,9,b,10,ni,13,j,14)	C,_t,i,j,A,tmp	C	<0>	
15					}
16					

Each statement can potentially have its own mapping between processor indexes and values of `dist` variables. When the debugger is invoked, it computes this mapping by expanding the `rep` and `dists` of a program. For every statement with a PIL, we record the statement number from the β program, the value of the PIL, and the value of the `dist` variable. As an example, we compute the mapping for the statements in the block distributed program. Assume that there are only 2 processors; `_np` is equal to 2. We only need to know the value of `dist` variables, so we can ignore everything before line 7. Furthermore, the PIL for every statement is the same, so the mapping is the same for all statements. The mapping is as follows:

`dist` variable value	processor index
0	0
1	1

The mapping for the cyclically distributed program is the same.

After a processor is stopped because of an event, its virtual time is computed as follows. From the line number, we get a virtual time template. If the template has `dist` variables in it, their values are obtained from the table that maps processor indexes to `dist` variables. If the template has any loop variables in it, their values are obtained from the loop variables in the program.

As an example of computing the virtual time, if processor 1 stops at statement 13 in the cyclically distributed program, the virtual time template is (2,`k`,8,`_t`,9,`b`,10,`ni`,13,`j`,13). If the value of the loop counters are `k=3`, `_t=1`, `ni=1`, and `j=2`, then the current virtual time on processor 1 is (2,3,8,1,9,1,10,1,13,2,13).

6.4.2 Selecting the β time, setting breakpoints, examining and modifying variables, and reporting the current location

Before the debugger can decide if commands should be disallowed, it must compute the early operations. It must first choose t the β time, which is computed from the virtual time of each processor. The β time, along with the virtual time template is used to determine if each statement has been executed early. For the cyclically distributed program, if the virtual time of processor 0 is (2,3,8,1,9,0,10,1,13,5,13), and the virtual time of processor 1 is (2,3,8,1,9,1,10,1,13,2,13), then the β time is the minimum, or (2,3,8,1,9,0,10,1,13,5,13). For cyclic distributions in general, the β time is the time of the processor that has executed the least number of outer loop iterations. For the block distribution, it is the first processor that has not executed all of its iterations.

After computing the virtual time sets for each statement and processor, we conclude that statements 13 and 14 have early executions on processor 1. The rest of the section assumes that the program is in the state described above.

If the user wants to set a breakpoint on line 13, the debugger for distribution sets a breakpoint on every copy of that statement; there is one copy for each processor. The debugger for thread splitting must check if any of those statements are early. For our example, line 13 on processor 1 has an early execution, so the debugger must disallow this breakpoint. If the user sets a breakpoint on line 2, then the debugger for thread split-

ting would set a breakpoint on both copies. The thread splitting debugger determines that there are no early executions for any of the copies of line 2 and sets the breakpoints.

If the user examines the variable d, then the debugger for distribution inspects the copy of d that is local to the processor executing the current statement. That is processor 0. On processor 0, there are no early operations, thus there cannot be an early write of the variable d, so the thread splitting debugger does the read of the variable d on processor 0. If the user tries to modify the variable d, then the debugger for distribution modifies all copies of the variable. The debugger for thread splitting checks if there are any early reads or writes for the variable d for all copies. Processor 1 has early execution of statements 13 and 14, and from checking the table in FIGURE 6-3, it concludes that there is an early read of the variable d. It must disallow the modification of the variable d because one copy can't be modified.

If the user does a `where` command, the debugger for thread splitting passes back the current statement, which contains the statement number and processor index. The distribution debugger just peels off the processor index and returns the statement number. In the current state, the thread splitting debugger passes back line 13 on processor 0 as the current line and the distribution debugger passes back line 13 is the current statement. This is reported to the user as the current line.

6.4.3 Roll forward

Roll forward is used for three purposes. It is used to advance execution of a program so that there are no late operations, to rerun the program to a particular β time, and to run a program with a consistent breakpoint. The virtual time template and current virtual time are used to find the LUB for each processor, which is the point in the program at which it should stop.

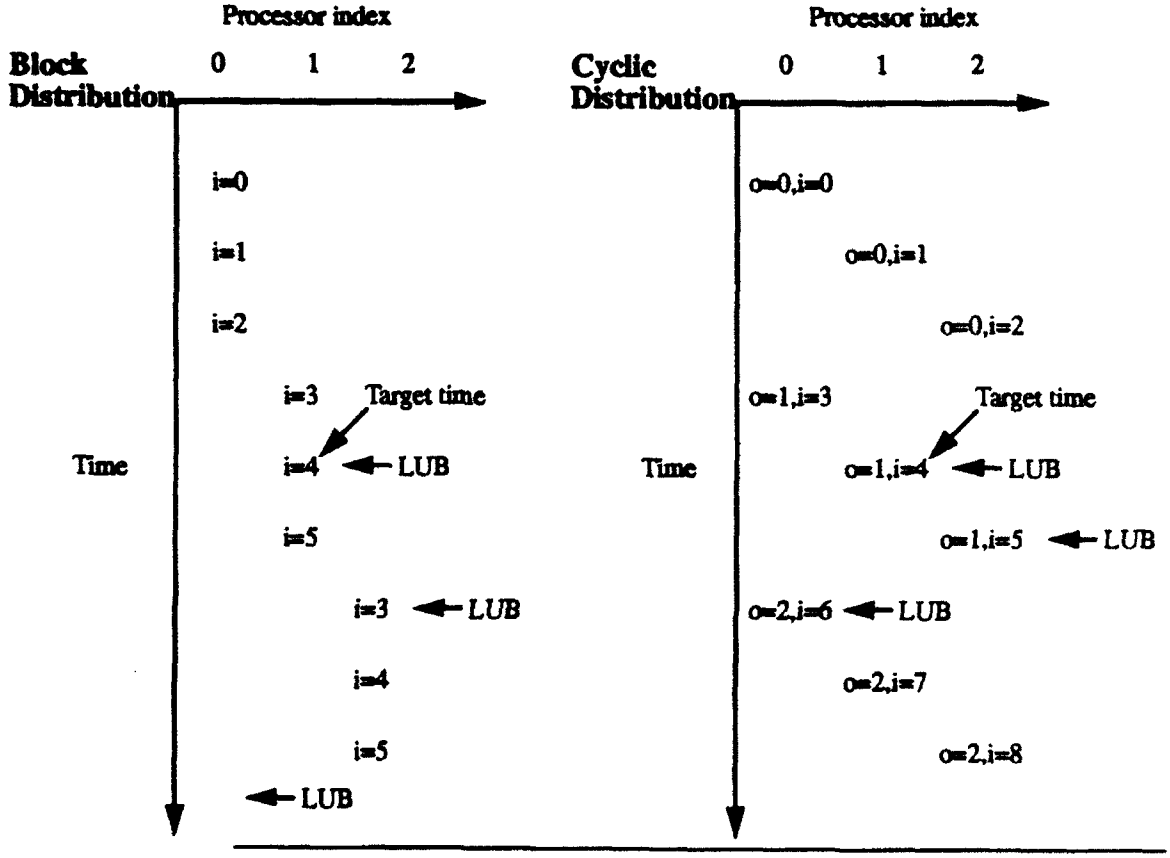
If the desired point in the block distributed program is `k=3`, `ni=1`, and `j=2` on statement 11 of processor 0, then if we substitute the values into the virtual time template of (2,k,7,b,8,ni,11,j,11) we have a virtual time of (2,3,7,0,8,1,11,2,11). This is also the LUB of processor 0. For processor 1, the LUB is at the beginning of its copy of the `i` loop. This is at statement 11, with a virtual time of (2,3,7,1,8,0,11,0,11).

We use FIGURE 6-4 to illustrate the general conditions for finding the LUB for each processor. We use a block distribution with a block size of 3 and a cyclic distribution where each processor gets 3 iterations. In the block distribution, for processors that execute iterations earlier than the target time, the LUB is the first thing after the loop; the processor executes its entire block. For processors that execute iterations after the target time, the LUB is the first iteration that it executes; the beginning of the block. For the cyclic distribution in the figure, `o` is the outer loop counter and `i` is the inner loop counter (the inner and outer loop are the result of the strip mine). The LUB for a processor is either the same or next iteration of the outer loop, depending on whether the processor comes before or after the processor executing the target time.

6.4.4 Consistent breakpoints

If the user sets a consistent breakpoint in a program, then we must compute the LUB for that breakpoint. FIGURE 6-5 illustrates where the LUB is for each processor before we

FIGURE 6-4 The position of the LUB for a given target time with the block and cyclic distribution



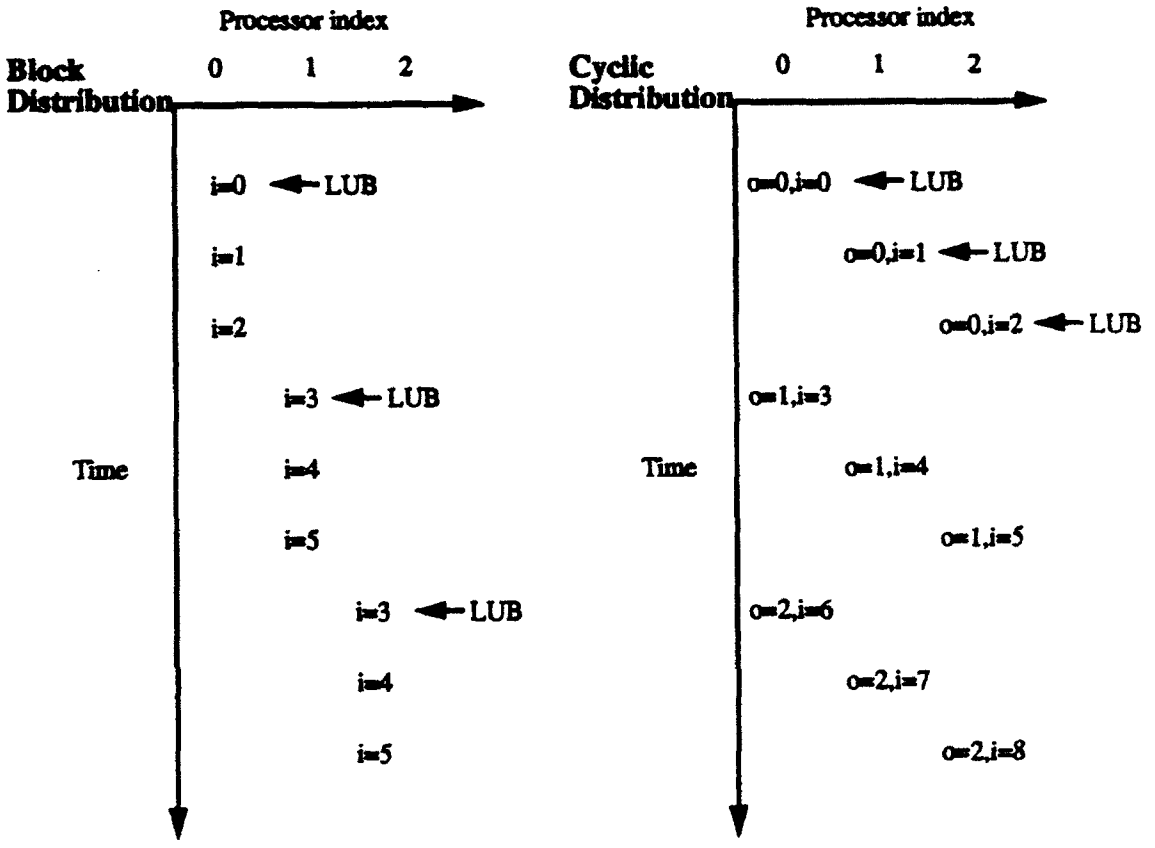
start execution of the loop. For the block distributed program, the LUB for each processor is the beginning of the block. If we reach a LUB for a processor without hitting a breakpoint first, then the new LUB for that processor is the end of the loop.

For a cyclic program, the LUB is always the next iteration that the processor executes. After we pass a LUB for a processor, the new LUB for that processor is the next iteration it executes.

6.5 Performance of block and cyclic distributions when debugging

We conclude this chapter by investigating some performance related issues of debugging for the block and cyclic distributions. The actual performance that a user sees is dependent upon the particular program, its communication, where the user decides to stop the program, and what variables are inspected or modified. However, it is possible

FIGURE 6-5 The position of the LUB for consistent breakpoints with block and cyclic distributions



to make some simple statements about the relative performance of the loop distribution models. For the following analysis, we assume that all processors complete loop bodies at the same rate. This is true if there is no synchronization in the loop body, and the work in each iteration is data independent.

6.5.1 Early operations

The more early operations there are, the more likely it is that we would have to disallow a command. The block and cyclic distributions can be expected to have different behavior in terms of the number of early operations.

We count the number of early loop iterations to compare which distribution has more early operations. For both distributions, the β time is the earliest virtual time of all the processors. Under the equal progress assumption, for a block distributed loop the earliest virtual time is always on the first processor. This is because all of the iterations on the first processor have virtual times less than the iterations of all the other processors.

Hence, all the iterations executed by all the processors except for the first one are early. This is $s(p-1)$ where s is the number of iterations executed on one processors and p is the number of processors. For the cyclic distribution, iterations are assigned in round robin fashion. While iteration i is in progress on processor 0, iterations $i+1$ through $i+p-1$ are in progress on the other processors. All the iterations in progress on all but the first processor are early, but all other iterations already completed are not early. The number of early operations for the cyclic distribution is at most $p-1$. In comparison we would expect a loop that is block distributed to have many more early operations, hence it is more likely that commands would be disallowed when using this distribution.

6.5.2 Parallelism during roll-forward

When we want to stop the program at a particular virtual time, for example we are rolling forward a set of processors, then we would like to execute the program with the maximum parallelism available so that we may reach that point as quickly as possible. If we have a block distribution and we want to stop at virtual time i, then we can allow all processors with an index less than $\lfloor i/b \rfloor$ to execute their complete loops, while processors with an index greater than $\lceil i/b \rceil$ cannot not execute any iterations. The degree of parallelism available is thus determined by the desired loop iteration; the higher the loop iteration the more parallelism that can be exploited.

For the cyclic distribution, since iterations are assigned to processors in a round robin fashion, we can allow all of the processors to execute the first $\lfloor i/p \rfloor$ iterations that they are assigned. The remainder, which is $i - \lfloor i/p \rfloor p$ are executed by processors 0 through $i - \lfloor i/p \rfloor p - 1$.

Thus, a cyclic distribution will reach the target point faster because it can exploit more parallelism. If we want to execute the first i iterations of the source loop, then the block distribution will take $\min(n/p, i)$ steps, where n is the total number of iterations to be executed. The cyclic distribution will take $\left\lfloor \frac{i}{p} \right\rfloor$ steps to complete the first $\left\lfloor \frac{i}{p} \right\rfloor p$ iterations and 1 more step to complete the remainder, which is $i - p\left\lfloor \frac{i}{p} \right\rfloor$. The total is thus $\left\lfloor \frac{i}{p} \right\rfloor + 1$.

6.5.3 Synchronization overhead of consistent breakpoints

When we execute a block or distributed loop with a consistent breakpoint set, we cannot take advantage of any parallelism, because the breakpoint may stop at any iteration. If we want to stop in a state without any early iterations, we can only execute one iteration at a time.

The execution of a loop with a consistent breakpoint should be slower than the sequential version of the loop because extra synchronization must be done. Synchronization occurs whenever a program reaches its LUB; a breakpoint occurs and the processor waits for the debugger to let it go forward. The debugger lets it go forward when all the iterations with lower virtual times complete. As described in Section 6.4.4, each proces-

sor of the block distributed program must synchronize before executing any of its block of iterations. Once it has passed its LUB it does not need to synchronize again until the end of the loop. The cyclic distribution must synchronize before every iteration.

The block distribution can be expected to synchronize less than the cyclic distribution; hence consistent breakpoints will be faster for the block distribution. For the block distribution, if the real breakpoint occurs in iteration i, then the number of synchronization points is $\left\lfloor \frac{ip}{n} \right\rfloor$. For a cyclic distribution, the number of synchronization points is i.

6.6 Summary

We have presented examples of compilation and debugger functions for programs with block and cyclic distributions. We also examined some performance issues related to debugging. The cyclic distribution can be expected to have less early operations and hence is less likely to disallow a command. It is also able to exploit more parallelism when rolling forward to a particular virtual time. However the block distribution can be expected to have less overhead when executing with a consistent breakpoint.

CHAPTER 7

Limitations of thread splitting

In this chapter, we identify the types of parallelizing transformations for which it is possible to construct a debugger using the thread splitting methodology. The main characteristic of a transformation that determines if it is suitable is the type of assignment it uses to map operations in the source program to processors. An assignment of a sequential program is specified by giving each operation a processor index label (PIL). The PIL is the processor that executes the operation in the parallel program. We call an assignment of operations to processors *data independent* when the PIL's are constant. We call an assignment of operations to processors with non-constant PIL's a *data dependent* assignment. Thread splitting only allows data independent assignments because PIL's must be constant.

A data independent assignment is desirable because it makes it possible for the debugger to take advantage of static program information, which is important for efficiency. However, most parallelizing transformations use data dependent assignments; a practical compiler and debugger must be able to support both.

If a transformation uses a data dependent assignment, source programs for the transformation can be rewritten so that a data independent assignment can be used to achieve the desired mapping of operations to processors. Thus, a debugger need only allow data independent assignments. However, a program generated in this way may be less efficient than a program generated from a data dependent assignment. For transformations where the assignment is data dependent but known at compile-time, the loss in efficiency is small. This includes the block, cyclic, and block-cyclic distributions. For transformations where the distribution of operations to processor is dynamic, the loss in efficiency can be significant. This includes user mapped and dynamically load balanced distributions.

In Section 7.1, we define data independent and data dependent assignments and describe their relationship to thread splitting and debuggers. Section 7.2 explains why a program generated with a data independent assignment can be less efficient than a program generated with a data dependent assignment. Section 7.3 shows how to rewrite a program so that a data independent assignment can be used. Section 7.4 describes how to bound the overhead for an assignment of a transformation and analyzes the commonly used distributions.

7.1 Data dependent and independent assignments

Generating a parallel program from the sequential one can be modeled as an *assignment* of operations to processors. The assignment can be thought of as the specification of the distribution. Alternatively, a distribution can be thought of as an implementation of an assignment.

An assignment is *data independent* if, for all executions, an operation is always executed on every member of the same set of processors, even if the operation is executed more than once. An operation may be executed more than once if it is inside a loop. A data independent assignment is shown in FIGURE 7-1. All iterations of the first loop are always executed on processor 0, all iterations of the second loop are always executed on processor 1, and all iterations of the last loop are always executed on processor 2.

FIGURE 7-1 Examples of data dependent and data independent schedules

data independent assignment

```
<0>   for (i = 0; i < n/3; i++)
<0>      b = 1;
<1>   for (i = n/3; i < 2*n/3; i++)
<1>      b = 1;
<2>   for (i = 2*n/3; i < n; i++)
<2>      b = 1;
```

data dependent assignment

```
<i/3>    for (i = 0; i < n; i++)
<i/3>       b = 1;
```

An assignment is *data dependent* if the function that assigns operations to processors is dependent on data in the program. An example can be found in FIGURE 7-1. The result of dividing the loop counter by 3 is the index of the processor that executes the iteration. In general, any distribution of loop iterations (e.g. block and cyclic) requires a data

dependent assignment. Thread splitting cannot be used directly when a data dependent assignment is desired because only constant PIL's are allowed.

A data independent assignment is desirable because it makes it possible for the debugger to take advantage of static information about the assignment of operations to processors in a program. Static information is useful for mechanisms related to virtual time such as computing early operations, rolling forward a program, and consistent breakpoints. If the debugger does not take advantage of static information, then it would not be practical to provide some of these mechanisms.

A simple example of how a debugger can take advantage of static information can be found in FIGURE 7-1. If we use a data independent assignment as is shown in the top of the figure, and the current statement is selected to be the third statement (the `for` loop), then it is clear that processor 0 should have executed all of its iterations and processor 2 should not have executed any iterations if we want the current state to be consistent with a state of the β program. By contrast, if we have the data dependent assignment found in the bottom of the figure and the current line is chosen to be the second line of that program, then the debugger must know something about the behavior of the function in the PIL's and the variables they reference to determine which iterations each processor should execute.

7.2 Why programs with data independent assignments can be less efficient

The execution overhead introduced by rewriting a program so that a data independent assignment can be used is inherent in how much is known at compile-time about the assignment. If the assignment is known at compile-time, then the overhead is low. If there are points in execution of the program where the assignment can only be resolved at run-time, there is some overhead. The total overhead depends on how often the assignment must be resolved at run-time.

To illustrate the differences between data dependent and data independent, we describe an example where the assignment cannot be known at compile-time. We describe how to write the program so that a data dependent assignment can be used, then show how a data dependent version can be more efficient.

If we are allowed to use a data dependent assignment, we can simply give each operation in the β program a PIL that computes the processor which should execute it. In the ω program, which is generated from the β program, only the processor that is assigned the operation executes it.

An example of a data dependent assignment is found in FIGURE 7-2. In this program, a user-defined map is used to determine which processor executes each iteration; iteration i of the loop is executed on processor `map[i]`. Because the array `map` is set at run-time, any processor can execute any iteration. Assume that at run-time, before executing the loop, the program computes the set of iterations that each processor must execute. The result is stored in two arrays—`its` and `umap`. For processor p, `its[p]` is the number of iterations executed by that processor and {`umap[p][0]`, `umap[p][1]`, ...,

`umap[p][its[p]-1]` is the set of iterations that processor p executes. In the program, each processor just executes its assigned set of iterations according to the `its` and `umap` arrays.

The PIL for each statement uses the `map` array to specify which processor executes the iteration. Some statements have a PIL of `map[i]` and others have a PIL of `map[i+1]` because the variable `i` is incremented in the middle of the loop. In the ω program, each processor only executes its assigned iterations and does not execute the loop control for any other iterations.

The β program that must be used for a data independent assignment is very different for this example. In general, if there is a point m in the control flow of the β program where any one of a set of processors s can be assigned the same computation c, then for every processor p in s, there must be one execution path starting at m that has a copy of the computation c with a PIL of p. A conditional in the program decides at run-time which path to execute, in effect deciding which processor should execute the computation c. In the ω program, which is extracted from the β program, every processor in s must execute the conditional. The processor that is assigned the current instance of c executes the path which contains c as well.

This is illustrated by the program with a data independent assignment in FIGURE 7-2. As before, a user-defined map is used to determine which processor executes each iteration. Because any processor can execute any iteration, there must be one execution path through the loop body of the β program for each processor. In the ω program in the figure, every processor executes the loop control once for every iteration of the β program and a conditional decides if the processor executes that particular iteration. By comparison, a program where a data dependent assignment can be used is much more efficient because each processor only executes the loop control for its assigned iterations.

Overhead is any work present in a program with a data independent assignment that need not be present in a version of the program with a data dependent assignment. For a particular data independent assignment, the overhead the program incurs is dependent on the way the β program is written. If we know at compile-time how work is assigned to processors, then we can change the structure of the program so that we can reduce the number of processors that can potentially execute the next operation. By doing this, we reduce the overhead of the program. If the assignment is known completely at compile-time, then there is no overhead because we can construct a program where only one processor executes the next iteration.

The example in FIGURE 7-3 illustrates the way we can exploit compile-time knowledge of the assignment to reduce overhead. The program on top is the α program; below are two possible β and ω programs for the same assignment. For the assignment, iteration i of the α loop is executed on processor $i/3$. We assume that there are 3 processors in the example. In the first β program, there is one execution path through the loop body for each possible assignment of an iteration of the α loop. That is, if an iteration is assigned to processor 0, the body of the first `if` is executed, if an iteration is assigned to processor 1, the body of the second `if` is executed, and so on. In the ω program, every processor executes the loop control for every iteration as well as its assigned iterations. In the second β program, we have created three copies of the α loop. Because there are

FIGURE 7-2 Examples of data dependent and data independent schedules

data dependent assignment

β program

```
<0>,<1>,<2> i = -1;
<map[i+1]>  for (t = 0; t < its[_id]; t++){
<map[i+1]>      i = umap[_id][t];
<map[i]>        b = 1;
            }
```

ω program

```
i = -1;
for (t = 0; t < its[_id]; t++){
   i = umap[_id][t];
   b = 1;
}
```

data independent assignment

β program

```
<0>,<1>,<2> for (i = 0; i < 2; i++) {
<0>             if (map[i] == _id)
<0>                b = 1;
<1>             if (map[i] == _id)
<1>                b = 1;
<2>             if (map[i] == _id)
<2>                b = 1;
            }
```

ω program

```
for (i = 0; i < 2; i++) {
   if (map[i] == _id)
      b = 1;
}
```

multiple copies of the loop, at every point in the execution of the β program, only one processor can execute the next iteration of the α program. In the ω program every processor only executes its assigned iterations.

FIGURE 7-3 Two possible ways of writing a β program for the same assignment

α program

```
for (i = 0; i < n; i++)
   b = 1;
```

β program

```
<0>,<1>,<2> for (i = 0; i < n; i++) {
<0>            if (_id == i/3)
<0>               b = 1;
<1>            if (_id == i/3)
<1>               b = 1;
<2>            if (_id == i/3)
<2>               b = 1;
            }
```

ω program

```
for (i = 0; i < n; i++) {
   if (_id == i/3)
      b = 1;
}
```

β program

```
<0>   for (i = _id*n/3; i < (_id+1)n/3; i++)
<0>      b = 1;
<1>   for (i = _id*n/3; i < (_id+1)n/3; i++)
<1>      b = 1;
<2>   for (i = _id*n/3; i < (_id+1)n/3; i++)
<2>      b = 1;
```

ω program

```
for ( i = _id*n/3;
      i < (_id+1)n/3;
      i++)
   b = 1;
```

7.3 Generating a β program

In this section, we describe how to structure a β program to minimize overhead when a data independent assignment is used. We want to be able to describe the assignment of a transformation independent of the particular program to which it is applied. For this reason, we use a simple model of a program as a single loop with a single statement body. The iterations of the loop are mapped to a 1 dimensional processor array.

For distribution, the compiler inputs an α program, and uses a combination of conditionals, loops, and sequences of code to produce a β program with a data independent assignment such that each processor executes the appropriate iteration of the loop from the α program. Each construct is useful for different patterns in the assignment.

If the assignment follows some finite sequence that is fixed at compile-time, then we can create multiple copies of the loop body of the α program as a sequence and assign each copy to the appropriate processor. An example is shown in part (a) of FIGURE 7-4. The pattern is (0,2,0,1,1,2) so we create 6 copies of the loop body and give them PILs from the sequence. If the assignment repetitively assigns an iteration to the same processor, then we can put the body of the α program in a loop and give it the PIL of the processor. This is illustrated in part (b) of FIGURE 7-4 where a data dependent number of iterations are assigned to processor 0. If at some point in the program, the assignment of an iteration can only be determined at run-time, then a conditional can be used to select which processor executes the next iteration. In part (c) of FIGURE 7-4, iterations can either be mapped to processor 0 or processor 1. The conditional selects which processor executes the iteration.

Each of the above constructs can be combined. For example, if the assignment repeatedly assigns iterations to the same sequence of processors, then we replicate the body according to the sequence and place that inside a loop.

7.4 Estimating the overhead of data independent assignments

The overhead of using a data independent assignment depends on the parallelizing transformation, the program, and the data the program is executed on. To determine if a parallelizing transformation is suitable for thread splitting, we would like to bound the extra work for all programs generated by a parallelizing transformation.

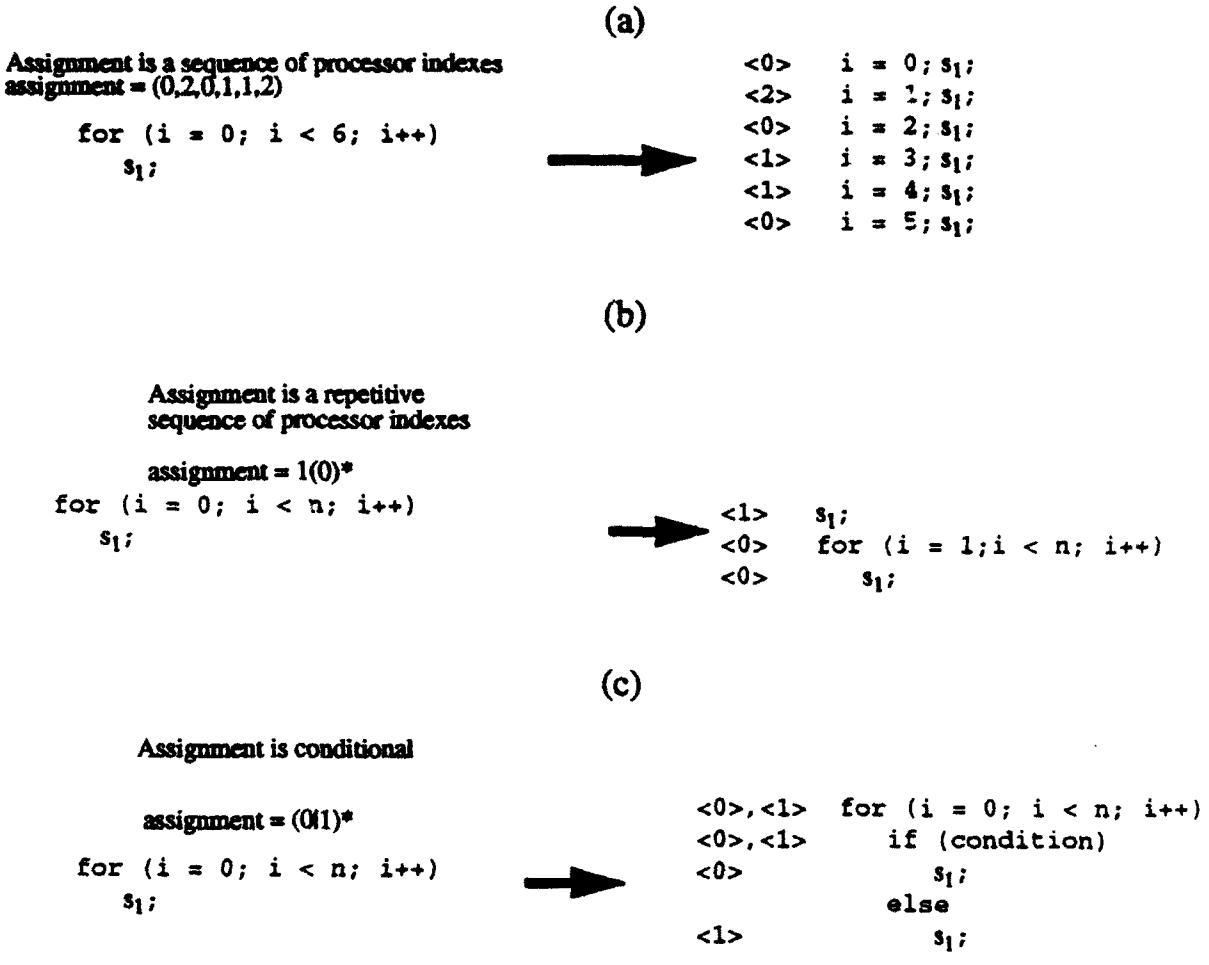
It is not important how an assignment decides how to map iterations to processors; our main concern is the set of possible assignments for all programs. For this reason, we describe the result of assignment without specifying how it is done. A *schedule* of a loop represents the assignment of iterations to processors for a single execution of the program. A program can have different schedules if the program is executed on different data. The schedule can be represented by a string of numbers $p_1, p_2, \ldots, p_n$ where iteration i is executed on processor p_i. For the estimation of overhead, we assume that all programs terminate, thus schedules are finite.

For each parallelizing transformation, there is a set of schedules possible for *all* executions of the single loop program. The set of schedules can be specified with a regular expression, where every schedule possible for that transformation is contained in the set of strings defined by the regular expression.

To denote a regular expression, we use numbers, which represent processor indexes, parentheses, and the operators | and *. The operator | is used for selection of alternates. If there is a regular expression (a | b), then the set of strings in the language is a and b. The operator * (also called closure) is used for repetition 0 or more times. The expression a(b*) has the following strings in its language: a, ab, abb, abbb ....

A regular expression was chosen to represent the set of schedules for several reasons. First and most important, there is a straightforward method for placing an upper bound

FIGURE 7-4 Converting loops with data dependent PILs into loops with data independent PILs



on the overhead of a parallelizing transformation given the regular expression defining the set of schedules. This is explained in detail in Section 7.4.2.

Second, a regular expression is always sufficient to describe a superset of the set of schedules for a program. While we may not always be able to write a regular expression that includes exactly the set of schedules possible for a parallelizing transformation, we can always find one that includes every possible schedule. That is, there may be additional strings in the language that are not schedules of the program. For example, the set of schedules for a program where statements are only mapped to processors 0 and 1 is

always contained in (0|1)* so this regular expression can be used to describe any set of schedules.

Third, the regular expression is simple to write. The regular expression defining the set of schedules captures the information necessary to measure the overhead, but leaves out most of the details that are irrelevant. Writing the regular expression is much easier than writing the β program. In Section 7.4.1, we list all the regular expressions for the common distributions. They are short and have simple structures.

Fourth, the structure of the regular expression suggests a structure for implementing an efficient β program. To generate a β program, every number can be replaced with a copy of the body with the number as a PIL, every |-clause can be replaced with a conditional, and every closure can be replaced with a loop. The regular expression does not contain enough information to generate a complete β program because it is missing the conditions for assignment to a particular processor. That is, if we had (0 | 1) we know that the iteration is assigned to processor 0 or 1, but we do not know when it is assigned to each processor.

7.4.1 Regular expressions for transformations

In this section we give the regular expressions for the common parallelizing transfc.mations. The four distributions employed by most parallelizing compilers are block, cyclic, block-cyclic, and user mapped.

In the block distribution, contiguous blocks of iterations are divided among processors. The general form for a 1 dimensional processor array where processor indexes range from 0 to *n* is (0*1*2*... *n**). A schedule is some number of iterations executed on processor 0, followed by some iterations executed on processor 1, and so on. The block size is typically the same for all processors. However, depending on the program, some processors on either end may not be assigned any iterations and the first processors on either end that are assigned iterations may only execute a partial block. For example, if there is a block size of 4, then some of the schedules that are possible for an array of length 3 are: 000011112222, 11112222, and 00111122. The regular expression is general enough to express all of these schedules. It also includes some schedules that are not possible for a block distribution, such as one that skips a processor when assigning iterations (e.g. 0022). Including the extra schedules in the set does not increase the potential overhead, as is shown later.

For a cyclic distribution, iterations are assigned in a repeating pattern. The general form is (0 |)(1 |)(2 |)...(n |)(012...n)*(0 |)(1 |)(2 |)...(n |). The |-clause (*i* |) means that *i* may or may not appear in the schedule. A schedule can start and end anywhere in the repeating pattern. Some possible schedules for a three processor array are 012012012, 1201212, and 1201201201. The regular expression we chose to represent the set of schedules includes some schedules which cannot occur when using a cyclic distribution, such as 02012012 and 01201202.

The block-cyclic distribution is a combination of the two previous ones. The general form is 0*1*2*...n*(0*1*2*...n*)*0*1*2*...n*. Some possible schedules for a three processor array are 001122001122, 12200, and 00112200112.

For the mapped distribution, the assignment of iterations is determined by the contents of a mapping array which is set at run-time. Since nothing is known at compile-time, the regular expression defining the set of schedules must be very general. It has the form (0 | 1 | 2 | ... n)*. Any schedule is possible for this distribution. A dynamically load balanced distribution is one where the assignment of work to processors can be changed at run-time based on dynamic conditions. The regular expression describing the set of schedules for a dynamically load balanced program is the same as the mapped distribution because the schedule is not known at compile-time.

7.4.2 Computing the overhead from a regular expression

In this section, we describe how to determine if programs generated by a parallelizing transformation are expected to have high overhead. We first explain how to compute the upper bound of the overhead for a single execution of a program. We then make some assumptions about the expected behavior of programs to estimate the overhead for all programs under a single parallelizing transformation.

In the ideal case, each processor executes exactly the work that is assigned to it. In our model, the unit of work is a loop iteration. For every iteration a program executes, it must also execute the loop control once (updating the loop counter, checking it against the bounds); we count any additional loop control or any conditionals not contained in the loop body as overhead.

Given a regular expression defining the set of schedules for a program and a schedule of one execution of a program, we can bound from above the overhead of the program for the execution. The procedure is as follows. We have a pointer to a position in the schedule and a pointer to a position in the regular expression. Both pointers start at the beginning. At each step, we advance the pointer in the regular expression according to the current value pointed to in the schedule, and then advance the pointer in the schedule by 1 position. There are three constructs in the regular expression: sequences of symbols, |-clauses, and closures. We explain each of them below.

For a sequence, the next symbol of the schedule must match the next symbol of the regular expression. If they do not match, there is an error; this cannot happen if the regular expression includes the set of possible schedules. If there is a match, we advance the pointer in the regular expression by 1 position. When there is a |-clause in the regular expression, we pick one branch and advance the pointer to the beginning of it. If the next symbol in the schedule is the same as the first symbol in the left side of the |-clause, we move the pointer to the left side. If the next symbol of the |-clause is the same as the first symbol in the right side, we to move the pointer to the right side. If neither matches, there is an error. If both match, then we follow both paths in parallel and pick the one that ends without error and has the least overhead. Computing overhead is explained in the next paragraph. When the branch of a |-clause is finished, we advance the pointer in the regular expression to the first symbol after the |-clause. When there is a closure (*), we can either move the pointer in the regular expression to the first symbol after the end of the closure or move it to the first symbol in the closure. The decision about skipping to the end or moving to the beginning of the closure is made the same way as the decisions for the |-clause—we compare the next symbol in the schedule to the two possible next symbols in the regular expression. After the pointer is advanced beyond the last symbol in a closure, we move it to the beginning of the closure.

Overhead is computed as we step through the regular expression. Stepping through a sequence incurs no overhead. For a |-clause, every processor in *either* clause incurs one unit of overhead. For a closure, every processor contained anywhere in the closure incurs one unit of overhead for every repetition of the body of the closure. After finishing the schedule, we sum up the overhead for each processor and subtract off the number of iterations that were executed, because a program must execute the loop control once for every iteration. An overhead number is an indication of how much we can lose by using a data independent assignment. It is not an exact measure of time because we don't know how much of the overhead is executed in parallel and we don't know the relative cost in execution time of loop control versus the body of the loop.

The above method allows us to determine the overhead for one execution of one program. We really want to determine the expected overhead of a transformation on all executions of all programs. If we make some assumptions about the behavior of programs we can estimate the overhead. The two assumptions that we need to make are that loop iterations execute many times and that the number of processors is not larger than the number of iterations.

For the block distribution, the set of schedules is (0*1*2*... *n**). There can only be overhead if a processor in the sequence does not execute any iterations. Thus, the overhead can at most be equal to the number of processors and can only be significant if the number of processors is much higher than the number of iterations to be executed. This is not a common case, so the block distribution does not incur significant overhead.

For the cyclic distribution, the set of schedules is
(0 |)(1 |)(2 |)...(n |)(012...n)*(0 |)(1 |)(2 |)...(n |). The overhead occurs in the |-clauses which is at the beginning or end of a schedule. If the closure is executed a significant number of times, then its cost will dominate and the overhead will be negligible. Because we assume loops will execute many times, the overhead of the cyclic distribution is expected to be small.

The block-cyclic distribution is also expected to have low overhead because it is a combination of block and cyclic. Overhead is low when the pattern repeats many times and the number of processors is not much greater than the number of iterations, which is believed to be the common case.

The mapped distribution is an exception in that it has high overhead. The set of schedules is (0 | 1 | 2 | ... *n*)*; there is an overhead of 1 for each processor for every iteration of the loop. Unless the work of a loop body is very large, that is, much larger than the work it takes for every processor to execute the loop control for one iteration, the mapped distribution incurs too much overhead and cannot have an efficient data independent assignment. The same conclusions hold for a dynamically load balanced program.

7.4.3 Why the computed overhead is an upper bound

Our computation of the overhead for a schedule of a program is pessimistic; using a regular expression to measure overhead can only give us an upper bound. The answer is not exact because there can be more than one correct regular expression that we could use for a set of schedules, and each one can lead to a different estimate of overhead. How-

ever, we believe that choosing a regular expression that leads to an accurate answer is straightforward; for each of the distributions in the previous section, we used the most "natural" regular expression, and obtained the right answer. For the distributions with low overhead, it is only the beginning and end of the schedules that are not known at compile-time, the cost of the middle dominates and can be completely determined at compile-time. For the user mapped distribution, nothing is known at compile-time about its behavior, so it is simple to model accurately as well.

The overhead that we estimate from a regular expression is pessimistic for several reasons. First, the regular expression that we choose might include schedules that are not possible executions of a program. The unnecessary schedules might require more flexibility in the assignment that adds extra overhead. A trivial example of an overly general regular expression is $(0 \mid 1 \mid 2 \mid \ldots n)^*$ which includes all possible schedules for processors 0 through n, hence it could be the specification for any set of schedules. Using this regular expression as the specification of the set of schedules when we really have a block distribution would lead to the incorrect conclusion that there is significant overhead.

We might use a regular expression that includes extra schedules for two reasons. First, a regular expression has limitations on the languages that it can generate. For example, a regular expression can have a sequence repeat a fixed number of times or an unlimited number of times, but it is not possible to specify a regular expression where the number of repetitions is the same as the number of repetitions in another part of the string. An example where this applies to schedules is the block distribution, where the size of the blocks can change from program to program, but are constant across processors for one particular program. When we specify the set of schedules for a block distribution, we cannot require that the block sizes be the same across processors.

Another reason we might use an overly general regular expression is that it is too tedious to exactly specify the set of schedules. In practice, using overly general schedules is not a problem. For all but the mapped distribution, we have used an overly general schedule, but we still concluded that the overhead was low.

Even if we find a regular expression that exactly defines the set of schedules for a transformation, it is possible that there is another regular expression that defines the same set of schedules, but has a lower overhead. For example, the regular expressions $1^*1^*1^*1^*1^*1^*$ and 1^* define the same language, but the first one has more overhead for short schedules because we must skip over more closures.

7.5 Summary

In this chapter, we describe data independent and data dependent assignments of operations to processors. Thread splitting directly supports data independent assignments. If a transformation uses a data dependent assignment, the source program must first be rewritten so that a data independent assignment can be used. The conversion can add overhead to the program. We explain why this overhead occurs and describe how to construct a β program that minimizes the overhead. We then describe a method to measure the overhead for a single execution of a program, and use it to estimate the overhead for the common distributions. In general, if the assignment is known at compile-

time, then the overhead is low. In specific, the block, cyclic, and block-cyclic distributions have little overhead, while it is expected that the user mapped and dynamically load balanced distributions will have a great deal of overhead.

CHAPTER 8 Conclusions

Research in parallelizing compilers, especially for distributed memory machines, still has far to go before it will have the level of maturity found in compilers for sequential machines today. Debuggers for these compilers are an even newer topic of research. This thesis identifies some basic methods, but there is still much work to do in extending the types of transformations and source languages that can be debugged. We hope that good debugging tools will help move parallel programming into the mainstream.

This chapter summarizes some of the conclusions about debuggability of transformations in Section 8.1. In Section 8.2, we outline some conclusions concerning the construction of debuggers. The major contributions are listed in Section 8.3. In Section 8.4, some areas for future work are identified.

8.1 Debuggability of distributions

In CHAPTER 5, we showed that the common distribution models: block, cyclic, and block-cyclic can be debugged at the source-level. The following characteristics make it possible: the order of execution of operations on a single processor is preserved, and the values that are computed are the same in the source and target programs. The user mapped distribution also obeys these properties, but since the assignment of iterations to processors is not know at compile-time, it cannot be implemented efficiently with our method.

From a debugging viewpoint, the round-robin style scheduling of the cyclic distribution has two advantages over the block distribution. The benefits result from the fact that the order that work is completed in the parallel program is closer to the order that operations are executed in the sequential program. This is advantageous when the user interrupts the parallel program; there are less early operations. It is also better when trying to run the program so that it will stop at a particular virtual time; more parallelism can be

exploited to reach that time quickly. If consistent breakpoints are used, then the block distribution is superior. As shown in CHAPTER 6, the *round robin* scheduling of iterations of the cyclic distribution requires much more synchronization than the block distribution.

8.2 Building debuggers

In addition to identifying the types of transformations that can be debugged, our work has also made clear what information the debugger needs from the compiler and how debuggers can be structured to be independent of a particular parallelizing transformation.

First, the debugger must know the structural relationship between variables and lines in the source and target programs. This correspondence cannot be represented by a simple table look-up because the relationship is dynamic and may require a computation to be performed. An example of a dynamic case is when there are multiple copies of a loop counter. An example that requires computation is when an array is distributed and it requires computation to compute the processor index and local address.

Second, the debugger must know the source ordering of operations in the parallel program. From that information, the debugger can determine what order to present events such as breakpoints. It also uses the order to compute late and early operations which is then used to determine if a command should be disallowed. The source ordering can be specified with a program (the β program). If the β program has a data independent assignment, the debugger can take advantage of static information. However, if a dynamic distribution is necessary, then data dependent assignment must be supported efficiently by the debugger.

8.3 Contributions

Source-level debugging for automatically parallelized programs is a new area of study. Our work has identified the general problems that a debugger must solve and has provided solutions for some of the common parallelizing transformations.

First, we have identified the basic services that a source-level debugger must perform. They are structural mapping, which is necessary for sequential debugger as well, and dynamic order restoration, which is unique to MIMD execution. Second, we have developed a methodology that allows us to separate structural mapping from dynamic order restoration when constructing a debugger. This allows us to isolate the part of the debugger that manages parallelism in a module that is independent of the distribution. Third, we have developed techniques for implementing dynamic order restoration. This includes mechanisms for virtual time, handling out of order events, disallowing commands, and roll forward.

8.4 Future work

The work presented in this thesis can be extended in many ways. In this section, we discuss some of the areas that need future work.

8.4.1 Distribution models

The thread splitting model is sufficiently general to handle the common distributions where the assignment of operations to processors is known at compile-time. Some systems, particularly those that support dynamic load balancing, use more dynamic distributions that cannot be determined at compile-time. To efficiently support these types of distributions, thread splitting must be adapted to allow data dependent assignments. When the assignment is data independent, computing the virtual time and the set of late and early operations only requires information about the control flow of the program. Adding the capability to handle data dependent assignments requires that the compiler give more information to the debugger about the program behavior.

8.4.2 Source languages with explicit parallelism

Some examples of source languages with explicit parallelism are data parallel languages (including languages with vector operations) and languages with parallel loops. When programs with explicit parallelism are executed efficiently on MIMD machines, they can require dynamic order restoration. The techniques that are used for dynamic order restoration when the source language is sequential must be extended when the source language has explicit parallelism.

8.4.3 Roll back mechanisms

In the thesis, we assume that roll forward is inexpensive and that roll back is more costly. This is because roll back is implemented by re-executing the program from the beginning. If roll back were inexpensive, some of the basic methods for implementing dynamic order restoration might be different. As described, the debugger tries to provide truthful behavior in the presence of early operations. If roll back is inexpensive, the debugger can use it to eliminate early operations from a program state in the same way that the debugger can use roll forward to eliminate late operations.

Reversible execution[Pan 88][Tolmach 91][Feldman 88], which has been proposed as a method for debugging parallel programs, could be used to make roll back less costly. Interaction between the debugger and reverse execution facility presents some interesting possibilities. For example, a general reverse execution facility must be able to roll back to any previous state of the program. However, a source-level debugger only needs to roll back to a much smaller set of states. This information could permit the reverse execution facility to reduce the need for storage.

8.4.4 Computation of early operations

In the thesis, we describe conservative methods for computing the set of early operations. If we use more aggressive methods, we decrease the number of falsely labelled early operations. This decreases the chances of disallowing a command, which in turn may reduce the need for re-execution of the program.

We discuss two ways that the debugger can be more exact in the computation of early operations. First, if a conditional has already been executed, we assume that both branches of a conditional were executed. If a loop has already been executed, we assume that an infinite number of iterations were executed. This information is clearly conservative because a conditional only executes one branch and a loop only executes a finite set of iterations. If the debugger records dynamic information about the program while it is executing, for example branching information and loop counts, then it could more precisely compute the set of operations that were executed which would in turn allow the debugger to more precisely compute the set of early operations.

Second, if the debugger has precise information about which loop iterations are executed, it can use that information to determine which array elements the loop iterations have accessed. By default, the debugger must assume that all elements of an array are accessed if a loop body accesses an array. If the element that a loop iteration accesses is only dependent on the iteration number, then the debugger can determine the exact set of elements that have been accessed. Deciding if array subscripts are only dependent on iteration numbers requires data flow information such as the set of loop invariant variables and induction variables.

8.4.5 Performance debugging

Another interesting topic is source-level performance debugging. After the user has debugged the program, the next step is usually tuning. For parallel programs, tuning is especially important because speed is the motivation for using a parallel machine. Understanding the performance of a parallel program requires that the user understand how computation and data are distributed. However, it isn't necessary that the user know all of the details of the distribution and a source-level view while tuning the program can ease some of the programmer's burden. Many of the techniques presented in this thesis for relating sequential source and parallel target would also be useful for performance debugging in this context.

Bibliography

[Aho 86] Aho, A., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[Aral 88] Aral, Z. and Gertner, I. High-level debugging in Parasight. *Workshop on Parallel and Distributed Debugging*, pages 151-162. ACM, Madison, Wisconsin, May, 1988.

[Bacon 91] Bacon, D.F. and Goldstein, S.C. Hardware-assisted replay of multiprocessor programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 194-206. ACM, Santa Cruz, California, May, 1991.

[Bailey 88] Bailey, M.L., Socha, D., and Notkin, D. Debugging Parallel Programs using Graphical Views. *Proceedings of 1988 International Conference on Parallel Processing*, pages 46-49. IEEE, August, 1988.

[Baker 77] Baker, B.S. An algorithm for structuring programs. *Journal of the ACM.* 24(1):98-120, 1977.

[Balasundaram 89] Balasundaram, V., Kennedy, K., Kremer, U., McKinley, K., Subhlok, J. The ParaScope editor: an interactive parallel programming tool. *Proceedings of Supercomputing '89*, pages 540-550. Reno, NV, November, 1989.

[Bates 83] Bates, P. and Wileder, J. C. An Approach to High-Level Debugging of Distributed Systems. Johnson, M. S. (editor), *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 107-111. ACM, Pacific Grove, California, August, 1983.

[Booth 86] Booth, M. and Misegades, K. Microtasking: a new way to harness multiprocessors. *Cray Channels.* 8(2):24-27, Summer, 1986.

[Brooks 92] Brooks, G., Hansen, G. J., and Simmons, S. A new approach to debugging optimized code. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 1-11. ACM, San Francisco, California, June, 1992.

[Bruegge 91] Bruegge, B. A portable platform for distributed event environments. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 184-193. ACM, Santa Cruz, California, May, 1991.

[Callahan 90] Callahan, D., Kennedy, K., and Subhlok, J. Analysis of event synchronization in a parallel programming to. *Second ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, pages 21-30. ACM, Seattle, WA, March, 1990.

[Cft 90] *CF77 Compiling System, Volume 1: Fortran Reference Manual* Cray Research, Inc., 1990.

[Chatterjee 91] Chatterjee, S. *Compiling data-parallel programs for efficient execution on shared-memory multiprocessors.* PhD thesis, Carnegie Mellon University, October, 1991.

[Cohn 91] Cohn, R. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 132-143. Santa Cruz, California, May, 1991.

[Copperman 90] Copperman, M.and McDowell, C. E. *Detecting Unexpected Data Values in Optimized Code.* Technical Report 90-56, Computer Research Laboratory, University of California at Santa Cruz, October, 1990.

[Coutant 88] Coutant, D. S., Meloy, S., and Ruscetta, M. DOC: A practical approach to source-level debugging of globally optimized code. *Proceedings of the 1988 Conference on Programming Language Design and Implementation*, pages 125-134. Atlanta, Georgia, June, 1988.

[Cytron 90] Cytron, R., Lipkis, J., Schonberg, E. *Proceedings of Supercomputing '90*, pages 398-406. New York, NY, November, 1990.

[Dinning 91] Dinning, A. and Schonberg, E. Detecting access anomalies in programs with critical sections. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85-96. ACM, Santa Cruz, California, May, 1991.

[Emrath 89] Emrath, P.A., Ghosh, S., and Padua, D.A. *Proceedings of Supercomputing '89*, pages 580-588. Reno, NV, November, 1989.

[Feiler 82] Feiler, P.H. *A Language-Oriented Interactive Programming Environment Based on Compilation Technology.* PhD thesis, Carnegie-Mellon University, May, 1982.

[Feldman 88] Feldman, S. and Brown, C. IGOR: A System for Program Debugging via Reversible Execution. *Workshop on Parallel and Distributed Debugging*, pages 112-123. ACM, 1988.

[Forin 88] Forin, A. Debugging of heterogeneous parallel systems. *Workshop on Parallel and Distributed Debugging*, pages 130-140. ACM, Madison, Wisconsin, May, 1988.

[Fortrand 92] Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C., Wu, M. FORTRAN D Language Specification. 1992.

[Francioni 91] Francioni, J., Albright, L., and Jackson, J. Debugging parallel programs with sound. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 68-75. ACM, Santa Cruz, California, May, 1991.

[Gupta 88] Gupta, R. Debugging code reorganized by a trace scheduling compiler. Kartashev, L.P. and Kartashev, S.I.(editors), *Supercomputing '88*. International Supercomputing Institute, Inc., 1988.

[Gupta 92] Gupta, M. and Banerjee, P. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*. 3(2):179-193, March, 1992.

[Heath 91] Heath, M.T. and Etheridge, J.A. Visualizing the performance of parallel programs. *IEEE Software*. 8(5):29-39, September, 1991.

[Hennessy 82] Hennessy, J. L. Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems*. 4(3):323-344, July, 1982.

[Jefferson 85] Jefferson, D. Virtual Time. *ACM Transactions on Programming Languages and Systems*. 7(3):404-425, July, 1985.

[Jefferson 87] Jefferson, D. and others. *Distributed Simulation and the Time Warp Operating System*. Technical Report TR CSD-870042, Department of Computer Science, UCLA, 1987.

[Knobe 90] Knobe, K. and Lukas, J., and Steele, G. Data Optimization: Allocation of Arrays to Reduce Communication of SIMD Machines. *Journal of Parallel and Distributed Computing*. 8102-118, February, 1990.

[Leblanc 87] LeBlanc, T. J. and Mellor-Crummey, J. M. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*. C-36(4):471-482, April, 1987.

[Li 91] Li, J. and Chen, M. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and distributed computing*. 13(2):213-221, October, 1991.

[Mehrotra 90] Mehrotra, P. and Van Rosendale, J. *Programming distributed memory architectures using Kali*. Technical Report ICASE Report No. 90-69, Institute for Computer Application in Science and Engineering, October, 1990.

[Miller 88] Miller, B.P. and Choi, J.-D. A mechanism for efficient debugging of parallel programs. *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 22-24. ACM, Atlanta, GA, June, 1988.

[Netzer 91] Netzer, R.H. and Miller, B.P. Improving the accuracy of data race detection. *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 21-24. ACM, Williamsburg, VA, July, 1991.

[Padua 86] Padua, D. and Wolfe, M. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM.* 29(29):1184-1200, December, 1986.

[Pan 88] Pan, D.Z. and Linton, M.A. Supporting reverse execution of parallel programs. *Workshop on Parallel and Distributed Debugging*, pages 124-9. ACM, 1988.

[Pineo 91] Pineo, P. P. and Soffa, M. L. Debugging parallelized code using liberation techniques. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 108-119. Santa Cruz, California, May, 1991.

[Poly 89] Polychronopoulos, C.D., Girkar, M., Haghighat, M.R., Lee, C.L., Leung, B., and Schouten, D. Parafrase-2: an environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *International Journal of High Speed Computing.* 1(1):45-72, May, 1989.

[Ribas 90] Ribas, H. B. *Automatic Generation of Systolic Programs from Nested Loops.* PhD thesis, Carnegie Mellon University, June, 1990.

[Steele 84] Steele Jr, G.L. *Common Lisp.* Digital Press, 1984.

[Stevens 90] Stevens, K.G., Jr. and Sykora, R. The Cray Y-MP: a user's viewpoint. *COMPCON Spring '90*, pages 12-15. IEEE, San Francisco, CA, February, 1990.

[Sussman 91] Sussman, Alan. *Model-Driven Mapping of Computation onto Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, September, 1991.

[Tolmach 91] Tolmach, A.P. and Appel, A.W. Debuggable Concurrency Extensions for Standard ML. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 120-131. ACM, Santa Cruz, California, May, 1991.

[Tseng 89] Tseng, P. S. *A Parallelizing Compiler for Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, May, 1989.

[Wang 90] Wang, J.-Z. *Debugging parallel programs by trace analysis.* Technical Report UCSC-CRL-90-11, Computer Research Laboratory, University of California, Santa Cruz, March, 1990.

[Warren 78] Warren, H.S. and Schlaeppi, H.P. *Design of the FDS interactive debugging system.* Technical Report RC7214, IBM Research Report, July, 1978.

[Wholey 91] Wholey, S. *Automatic data mapping for distributed-memory parallel computers.* PhD thesis, Carnegie Mellon University. School of Computer Science, 1991.

[Zellweger 84] Zellweger, P. T. *Interactive Source-Level Debugging of Optimized Programs.* PhD thesis, University of California at Berkeley, May, 1984.

[Zernik 91] Zernik, D. and Rudolph, L. Animating work and time for debugging parallel programs - foundation and experience. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 46-56. ACM, Santa Cruz, California, May, 1991.