Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1990

Compiling Programs for Nonshared Memory Machines

Charles Koelbel

Report Number:

90-1037

Koelbel, Charles, "Compiling Programs for Nonshared Memory Machines" (1990). *Department of Computer Science Technical Reports.* Paper 38.
https://docs.lib.purdue.edu/cstech/38

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

COMPILING PROGRAMS FOR NONSHARED MEMORY MACHINES

Charles Koelbel

CSD-TR-1037
November 1990

Compiling Programs for Nonshared Memory Machines

Charles Koelbel

November 5, 1990

Abstract

Nonshared-memory parallel computers promise scalable performance for scientific computing needs. Unfortunately, these machines are now difficult to program because the message-passing languages available for them do not reflect the computational models used in designing algorithms. This introduces a semantic gap in the programming process which is difficult for the programmer to fill.

The purpose of this research is to show how nonshared-memory machines can be programmed at a higher level than is currently possible. We do this by developing techniques for compiling shared-memory programs for execution on those architectures. The heart of the compilation process is translating references to shared memory into explicit messages between processors. To do this, we first define a formal model for distributing data structures across processor memories. Several abstract results describing the messages needed to execute a program are immediately derived from this formalism. We then develop two distinct forms of analysis to translate these formulas into actual programs. Compile-time analysis is used when enough information is available to the compiler to completely characterize the data sent in the messages. This allows excellent code to be generated for a program. Run-time analysis produces code to examine data references while the program is running. This allows dynamic generation of messages and a correct implementation of the program. While the overhead of the run-time approach is higher than the compile-time approach, run-time analysis is applicable to any program. Performance data from an initial implementation show that both approaches are practical and produce code with acceptable efficiency.

Contents

1 Introduction 1

1.1 Nonshared Memory Computers . . . 1
1.2 Parallel Programming . . . 2
1.3 Research Goals . . . 6

2 The Kali Language 8

2.1 Kali Syntax and Semantics . . . 8
2.1.1 Sequential Constructs . . . 9
2.1.2 Processor Arrays . . . 9
2.1.3 Distribution Patterns . . . 10
2.1.4 Forall Statement . . . 15
2.2 Kali Implementation . . . 16
2.2.1 Basic Concepts . . . 16
2.2.2 Sequential Constructs . . . 17
2.2.3 Processor Arrays . . . 20
2.2.4 Distribution Patterns . . . 20
2.2.5 Forall Statement . . . 22

3 A Model for Data Distribution and Message Generation 25

3.1 The Structure of Generated Code . . . 25
3.2 Data Distribution . . . 27
3.2.1 The Mathematical Model . . . 28
3.2.2 One-Dimensional Distribution Patterns . . . 30
3.2.3 Multi-Dimensional Distribution Patterns . . . 30
3.2.4 Other Distributions . . . 32
3.3 Parallel Loops . . . 33
3.4 Communication Sets . . . 33
3.5 Iteration Sets . . . 36
3.6 Extensions to the Example . . . 37
3.7 Generating the Sets: Compile-time versus Run-time . . . 42

4 Compile-time Analysis 44

4.1 Notation . . . 45
4.2 Constant Subscripts . . . 46
4.3 Block Distributions with Linear Subscripts . . . 49
4.4 Cyclic Distributions with Linear Subscripts . . . 59
4.5 Extensions to Compile-time Analysis . . . 67

5 **Run-time Analysis** 69
5.1 The Inspector-Executor Strategy 69
5.1.1 The Inspector 69
5.1.2 The Executor 71
5.2 Representing the Communication and Iteration Sets 72
5.2.1 Iteration Sets 72
5.2.2 Communication Sets 73
5.3 A Practical Example: Unstructured Mesh Relaxation 75
5.4 Other Optimizations 78

6 **Experimental Results** 79
6.1 Methodology 79
6.2 Experiments with Compile-time Analysis 80
6.2.1 Absolute Overheads 80
6.2.2 Realistic Performance 81
6.2.3 Conclusions 87
6.3 Experiments with Run-time Analysis 87
6.3.1 Absolute Overheads 87
6.3.2 Realistic Performance 90
6.3.3 Conclusions 97

7 **Conclusions** 99
7.1 Summary of Contributions 99
7.2 Related Research 100
7.3 Directions for Future Research 102

ACKNOWLEDGMENTS

For their financial support at various times during my graduate student career, I gratefully acknowledge fellowships from AT&T and IBM, and research assistantships from the Institute for Computer Applications in Science and Engineering (ICASE) and the RAID distributed database project. When no research money was available, the Purdue CS department stepped in with teaching and grading assistantships. IBM and ICASE also allowed me to do on-site research at their facilities in Yorktown Heights, New York and Hampton, Virginia, respectively, for which I am duly grateful.

Compiler research cannot go forward without machines to run the programs, and so I am grateful to groups that have supplied me with facilities. Most prominent among the machines is bluecrab, the iPSC/2 at ICASE on which the performance measurements reported in Chapter 6 were made. I would like to thank Bob Voigt for providing access to that machine, and Tom Crockett and Scott Berryman for maintaining it. In addition, the NSF-sponsored Computing About Physical Objects (CAPO) project provided access to Purdue's NCUBE/7, which became the first machine to run a Kali program. Elias Houstis was most directly responsible for providing that access. The NCUBE was administered by Ko-Yang Wang, who also provided much helpful advice. The Kali compiler actually ran on various Sun workstations in Purdue's RAID lab and at ICASE, maintained by such people as Sean Hershberger, Dan Trinkle, Leon Clancy, and Tom Crockett.

Ideas do not form in a vacuum. Many of the concepts in Kali came from the BLAZE and E-BLAZE languages, which developed in part from discussions with Piyush Mehrotra, John Van Rosendale, Dennis Gannon, and Jan Cuny. Bob Voigt's support of research on Kali and its predecessors at ICASE was also a great help. The ideas for run-time analysis in Chapter 5 were sharpened considerably by talks with Joel Saltz, Scott Berryman, and Rik Littlefield. For general discussions and friendly support, thanks go to all the researchers at ICASE and the inhabitants of the RAID lab.

Andrew Royappa did a fine job of proofreading this thesis on short notice. I would also like to thank the members of my committee (Susanne Hambrusch, Tim Korb, Ryan Stansifer, and Piyush Mehrotra) for their careful reading and insightful comments. Piyush's comments in particular drastically improved the presentation of the first three chapters. Finally, I am indebted to Stephan Bechtolsheim and John Riedl, the creators of the thesis document style for LaTeX, and to Thomas Narten, Craig Wills, and Jameson Burt for their contributions to the final form of that style.

Chapter 1

Introduction

Nonshared memory parallel computers promise to provide high levels of performance scalable to large numbers of processors at a very modest cost. The lack of high-level support software has hampered their use, however. Until now, the only programming environments available on nonshared memory machines reflected the underlying architecture very directly. Unfortunately, this differs substantially from the model of computation that programmers use in designing programs. This situation leads to a semantic gap producing complex, inflexible programs. Our goal in this research is to provide higher-level languages for these machines to reduce this semantic gap. We first review the architecture and programming of nonshared memory machines, and then describe our approach.

1.1 Nonshared Memory Computers

Figure 1.1 shows a block diagram of a *nonshared memory computer*. It consists of a number of sequential processors, each with a section of memory which only it can access. The processors each operate independently of the others; that is, the system is a MIMD (*M*ultiple *I*nstruction stream, *M*ultiple *D*ata stream) machine in Flynn's terminology [Fly66]. If one processor needs data stored on another, the storing processor must explicitly send a message containing the data through the interconnection network and the first processor must receive that message. Examples of machines in this class include the Caltech Cosmic Cube [Sei85], SUPRENUM [Sol90], iPSC/2 [PL88] and NCUBE/7 [HMS+86].

Nonshared memory machines have two major advantages over competing parallel architectures: flexibility and scalability. The flexibility comes from the MIMD parallelism of the processors, which is difficult to simulate on SIMD (*S*ingle *I*nstruction stream, *M*ultiple *D*ata stream) machines such as the Connection Machine [TR88]. The highly modular connection between processors in a nonshared memory machine limits the bandwidth required of the interconnection network. Scalable networks that can achieve these bandwidths have been widely studied [Fen81]. This is in contrast to shared memory MIMD computers such as the BBN Butterfly [BBN87], IBM RP3 [PBG+85], and Sequent Balance [TGF87], which require very high bandwidths between processors and memory. Designing scalable interconnections for this situation has proven harder than in the nonshared memory case. This scalability has led researchers to predict very high performance for large nonshared memory machines. The Touchstone project, for example, expects to build a 2048-processor machine with a peak performance of over 150 billion floating-point operations per second [Lil90]. Such performance will make nonshared memory machines important in the area of high-speed computing for some time to come.

Three considerations are vital in efficiently programming a nonshared-memory machine.

1. The processor memories are small relative to the problems to be solved.

2. Communication is expensive relative to computation.

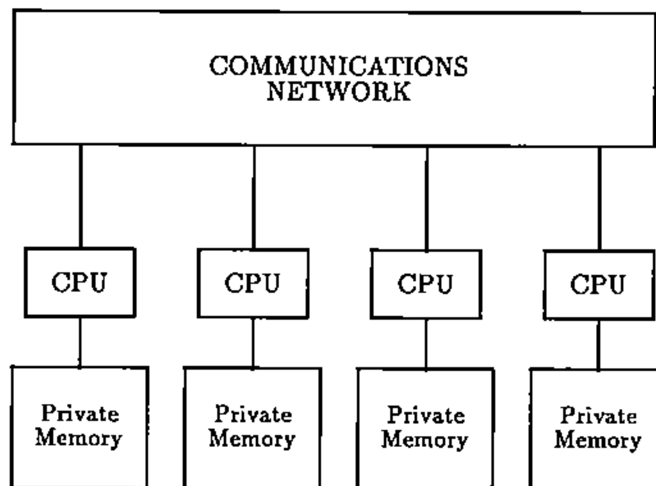


Figure 1.1: Nonshared memory parallel architecture

3. The message start-up time (latency) is high relative to the communications bandwidth.

The first consideration implies that large data structures such as arrays and matrices must be distributed among the processor memories. There are two reasons for this: data-intensive programs often need more memory than is available on a single processor, and coherency requirements mean data stored on many processors must also be updated everywhere it is stored. Replicated updates can add significant overhead to a program. For these reasons, large arrays are divided into sections with each processor "owning" one section. The communication overhead then implies that processors should do as much computation as possible on their own data. Thus, it is not enough to divide an array evenly; elements of the array which are used together in computations should also be stored on the same processor when possible. Different data distributions will imply different communications patterns and computation strategies. Choosing the best distribution will have a great impact on the efficiency of the program. The final consideration implies that small messages should be combined into larger ones where possible. These optimizations must be incorporated at some level in implementing any program on a nonshared memory machine to get reasonable performance. Because of their lack of pertainence to the algorithm, however, it would be best if they could be implemented automatically.

1.2 Parallel Programming

The most common model for designing parallel algorithms is the PRAM (*P*arallel *R*andom *A*ccess *M*emory) model [FW78]. In this model, a parallel computer consists of a group of independent processors which share a common memory. All accesses to this memory are considered equally fast. Many extensions to existing languages [Gel85, Han75, Jor86] and entirely new languages [MMS79, MV87, RSW90, Tse89] reflect this model directly. In all these languages, memory is essentially directly accessible to all processes, which conceptually map onto PRAM processors. Because of this underlying model, we refer to these languages as *shared-memory languages*. The languages provide constructs for creating and synchronizing processes, and often for protecting data from arbitrary modification. Many of these languages denote parallel execution by some form of parallel loops. These provide an iteration structure similar to sequential programs, except that iterations are executed concurrently on separate processors. The **forall** is a special case of this idea in which

```
var
   New_A, Old_A : array[ 0..N+1, 0..N+1 ] of real;

forall i in 1..N, j in 1..N do
   New_A[i,j] := 0.25 * ( Old_A[i-1,j] + Old_A[i+1,j] +
                 Old_A[i,j-1] + Old_A[i,j+1] );
end;
```

Figure 1.2: Jacobi iteration in a shared-memory language

the iterations may not have any data dependences on other iterations. (Data dependences within an iteration are allowed.) These conditions imply that no interprocessor synchronization is necessary during the **forall**. Figure 1.2 shows how a **forall** can be used in Jacobi iteration, a simple numerical algorithm.[1] In this program, N^2 processes are created, each of which computes one element of array *New_A*. The simplicity of this example points out the great advantage of these languages; because of their close relationship to theoretical models, they provide a high-level environment for the programmer.

The disadvantage of shared-memory languages on nonshared memory machines is the difficulty of mapping the PRAM model directly onto the underlying hardware. Instead, the most popular languages on these machines are *message-passing languages* [Ame83, GCKW79, Hoa78, INM86]. These languages provide an environment of asynchronously operating processes which interact by explicit messages. Direct sharing of data is forbidden. Such languages obviously reflect the underlying machine architecture very closely. This has advantages and drawbacks similar to assembly language on sequential computers. Because of the close relationship to the hardware, programs are very expressive, providing the fine control necessary for achieving optimal performance. On the other hand, the conceptual mismatch between the language and the programmer's conceptual model makes programs harder to write, debug, and maintain. In particular, in a message-passing language data distribution and message generation are the responsibilities of the programmer.

The effect of explicit data distribution and message generation on a program is clearly demonstrated by Figure 1.3, which implements the same program as Figure 1.2 in a message-passing language.[2] The program shown there is run on every processor of the nonshared memory machine, although some processors will not execute certain branches of the **if** statements. The added complexity is entirely due to data distribution, which forces the use of constants $low1$, $low2$, $high1$, and $high2$, and message-passing, which forces the **send** and **recv** statements. The complexity of the constant declarations is not due to an artificial data distribution; Figure 1.3 decomposes the original $(N+2) \times (N+2)$ matrices into contiguous $M \times M$ matrices as shown in Figure 1.4. (We also note that even this relatively complex program is not optimal. Jacobi iteration allows computation and communication to be overlapped, but Figure 1.3 does not attempt this. To do so would require additional **for** loops, which were omitted to save space.) Clearly, the level of detail involved in the message-passing version of Jacobi iteration will make the program more difficult to write and debug than the shared-memory version. Less obvious is the fact that the message-passing program is inflexible. In particular, changing the data distribution will change both the message-passing statements and the loop bounds. For example, Figure 1.5 implements Jacobi iteration using the skewed data distribution shown in Figure 1.6. Since the data distribution is orthogonal to the actual algorithm, it would be best if it could be specified separately at a high level. This is not possible with message-passing languages.

The disadvantages of writing software in message-passing languages have been a serious hindrance to the use of nonshared memory machines. A new approach to programming these architectures is needed to provide a higher-level environment for the user. Shared-memory languages, because they

[1] The program is written in BLAZE [MV87], a forerunner to the Kali language introduced in Chapter 2.
[2] The language used in this program and the next is BLAZE extended with message-passing constructs.

```
-- Code on processor proc[p1,p2], where proc has dimensions [0..P-1,0..P-1]

const
   M = (N+2) / P;          -- each processor stores an M by M array
   low1 = p1 * M;          -- lower bound for dimension 1 on this processor
   low2 = p2 * M;          -- lower bound for dimension 2 on this processor
   high1 = min( low1+M-1, N+1 );       -- upper bound for dimension 1
   high2 = min( low2+M-1, N+1 );       -- upper bound for dimension 2

var     -- note "border" elements added to array
   New_A, Old_A : array[ low1-1..high1+1, low2-1..high2+1 ] of real;

-- send data to other processors
if ( p1 > 0 ) then  send( Old_A[ low1, low2..high2 ], proc[p1-1,p2] );  end;
if ( p1 < P-1 ) then  send( Old_A[ high1, low2..high2 ], proc[p1+1,p2] );  end;
if ( p2 > 0 ) then  send( Old_A[ low1..high1, low2 ], proc[p1,p2-1] );  end;
if ( p2 < P-1 ) then  send( Old_A[ low1..high1, high2 ], proc[p1,p2+1] );  end;

-- receive data from other processors
if ( p1 > 0 ) then Old_A[low1-1,low2..high2] := recv( proc[p1-1,p2] ); end;
if ( p1 < P-1 ) then Old_A[high1+1,low2..high2] := recv( proc[p1+1,p2] ); end;
if ( p2 > 0 ) then Old_A[low1..high1,low2-1] := recv( proc[p1,p2-1] ); end;
if ( p2 < P-1 ) then Old_A[low1..high1,high2+1] := send( proc[p1,p2+1] ); end;

for i in max(1,low1)..min(N,high1), j in max(1,low2)..min(N,high2) do
   New_A[i,j] := 0.25 * ( Old_A[i-1,j] + Old_A[i+1,j] +
                 Old_A[i,j-1] + Old_A[i,j+1] );
end;
```

Figure 1.3: Jacobi iteration in a message-passing language, blocked distribution

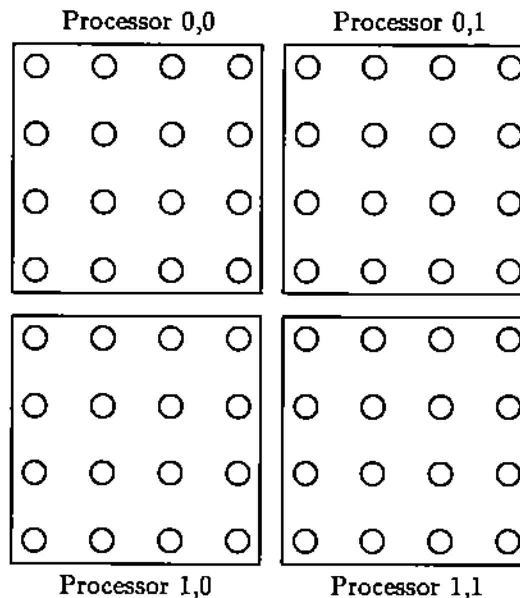


Figure 1.4: Data distribution used in Figure 1.3

```
-- Code on processor proc[p1], where proc has dimensions [0..P*P-1]

const
   P2 = P*P;     -- total number of processors
   M = (N+2) / P2;        -- each processor stores an (N+2) by M array
                          -- code below assumes P divides N+2 evenly

var
   New_A, Old_A : array[ 0..N+1, 0..M ] of real;
   temp1, temp2 : array[ 0..N+1, 0..M ] of real;          -- temporaries

-- send data to other processors
send( Old_A[ 1..N, 0..M ], proc[(p1-1)%P] );
send( Old_A[ 1..N, 0..M ], proc[(p1+1)%P] );

-- receive data from other processors
temp1[ 1..N, 0..M ] := recv( proc[(p1-1)%P] );
temp2[ 1..N, 1..M ] := recv( proc[(p1+1)%P] );

for i in 1..N
   var low_j, high_j : integer;
do
   if ( i%P2 = p1 ) then low_j = 1; else low_j := 0; end;
   if ( (i-N-1)%P2 = p1 ) then high_j := M-1; else high_j := M; end;
   for j in low_j .. high_j do
         New_A[i,j] := 0.25 * ( temp1[i-1,j] + temp2[i+1,j] +
                              temp2[i,j-1] + temp1[i,j+1] );
   end;
end;
```

Figure 1.5: Jacobi iteration in a message-passing language, skewed distribution

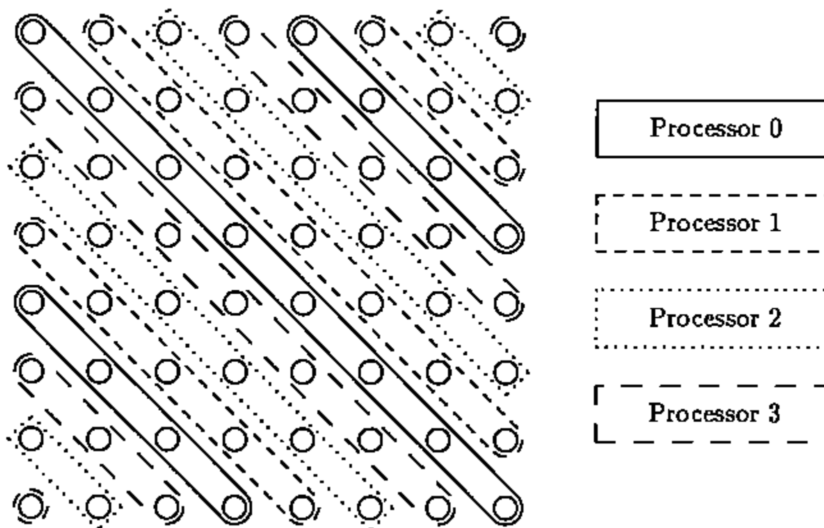


Figure 1.6: Data distribution used in Figure 1.5

are familiar to many users, seem to be a good candidate for providing such an environment. This is the approach that we advocate in the next section.

1.3 Research Goals

Our goal is to provide a high-level programming language for programming nonshared memory machines. Current message-passing languages for those machines, while they produce efficient code, are too difficult for programmers to use in the long run. Programs written in those languages are long, detailed, and inflexible even for simple tasks. Our approach is to instead offer languages closer to the PRAM model of computation with which the programmer is familiar. This will shift the burden of handling the details of implement data distribution and message generation from the programmer onto the compiler. Thus, the source programs will be shorter and simpler, making writing, debugging, and maintaining them easier.

Whether it is done by the compiler or by the programmer, the process of implementing a shared-memory program on a nonshared memory machine can be thought of as requiring five steps.

1. The data must be distributed across the processor memories in the nonshared-memory machine.

2. The computation must be divided among the processors. This is partially done in the original PRAM algorithm by the specification of parallelism; in a nonshared-memory machine, however, it is also necessary to identify the specific processor which performs each computation.

3. For each phase of the algorithm, every processor must receive the data it needs for its computation from the processor storing the data.

4. Because other processors are expecting data from it, each processor must also send messages to other processors.

5. During the computation, the processor must correctly access both its local data and the data received in messages.

We now consider each of these steps in more detail.

The reasons for distributing data were given in Section 1.1. Choosing a good data distribution is often the key to designing an efficient nonshared-memory program. This can be seen from the two versions of Jacobi iteration in Figures 1.3 and 1.5. In Figure 1.3, each processor (except edge elements) sends and receives 4 messages of size $O(N/P)$, while processors in Figure 1.5 each exchange 2 messages of size $O(N^2/P^2)$. Both programs have a computation time of $O(N^2/P^2)$. For moderate message start-up times, Figure 1.3 will have a much lower communication cost, as well as requiring only half the memory per node that Figure 1.5 does. This does not mean that blocked distribution is a panacea, however. In Gauss-Seidel iteration, an algorithm which solves the same problem as Jacobi iteration, parallelism appears as wavefronts sweeping diagonally across the matrix. In this situation, the blocked distribution will allow at most $O(P)$ processors to operate concurrently, while a skewed distribution allows $O(P^2)$. The message complexities of the two programs would be the same. Therefore, for Gauss-Seidel iteration the skewed distribution is preferred. In general, the problem of selecting an optimal data distribution is NP-complete or worse [Mac83]. Research is continuing into finding good heuristics, but at present choosing a data distribution requires insight into the program. Such insight is more often found in programmers than in compilers; therefore, our approach is to have the user provide a high-level description of the data distribution. This amounts to a short annotation accompanying array declarations. It is the compiler's responsibility to translate this annotation into the detailed declarations, addressing formulas, and code segments needed to allocate each processor's section of the array. In the future we hope to provide more compiler support for choosing the distribution.

As the last section pointed out, communication overhead on nonshared memory machines is quite high relative to the computation cost. This implies that computations must be distributed much like data. In the context of parallel loops, this translates into choosing a processor for each loop iteration. We use program annotations to specify this. The data distribution usually determines the computation distribution; a computation is performed either where its data is located or where its result will be stored. To accommodate this, our annotations for distributing the iterations of a **forall** can reference the distribution patterns of arrays. This relationship to data distributions suggests several heuristics for distributing loop iterations: execute on the processor storing the first data element referenced, or the first element assigned, and so on. We are continuing research on these heuristics.

Given the data distribution and the distribution of computations it is possible to find the set of data that each processor must reference. Any data that it does not already own must be received in a message and stored in a temporary buffer. The process of identifying these references and receiving the messages is often quite mechanical. Figures 1.3 and 1.5 show, however, that it can also require detailed calculations. Both of these reasons point to a job better suited to a computer than a human. Our approach therefore puts the burden of generating this code completely on the compiler. The programmer specifies no communication explicitly; instead, he or she merely references data in a logically shared memory. If a reference accesses nonlocal data, the compiler must translate that into efficient code for receiving the message.

The situation regarding sending messages is similar to that for receiving them: an often mechanical and always detailed task. We therefore put this burden on the compiler also. Programmers need not specify the source of data; it is the compiler's responsibility to determine that and arrange for it to be sent.

The sending and receiving of messages brings up a subtle point. Nonlocal data, although it is conceptually part of the same data structure, is not stored with the rest of that structure. Instead, it is usually stored in temporary locations. Sometimes, the temporaries may themselves be organized in rather complex ways unrelated to the main array. A single reference to a distributed array, therefore, may have several translations depending on where it was originally stored. The details of these translations will depend strongly on the exact methods used for communication, which in our approach are not available to the user. We therefore make it the compiler's responsibility to generate code to correctly access both local and nonlocal references.

To summarize, our goal is to automatically generate explicit message-passing code from a shared-memory language, given a distribution of data and **forall** iterations across processors. This translation can be thought of as a source-to-source translation from the shared-memory language to a message-passing language. The outline for reaching that goal is as follows. We first define the Kali language, a shared-memory language to be implemented on nonshared memory machines. This is done in Chapter 2. To formalize the task of implementing this language, we then develop a model of the process of data distribution on nonshared memory machines in Chapter 3. This model in turn will be used to derive two distinct methods of analysis applicable to Kali programs. Chapter 4 describes a compile-time analysis which produces very good code but does not apply to all programs. Chapter 5 describes a run-time analysis which has higher overheads but can be used for programs which are not amenable to compile-time analysis. Chapter 6 describes an implementation of both these forms of analysis and discusses the performance of each. Finally, Chapter 7 summarizes the research, discusses related work, and outlines future directions for this research.

Chapter 2

The Kali Language

Kali[1] is a language designed for expressing scientific computations on parallel, nonshared memory machines [MV89]. It contains the usual sequential constructs (assignment statements, for and while loops, if conditionals, etc.) along with constructs specifically designed for parallel execution. The design goals of Kali were to provide a high-level environment for programmers while still allowing efficient execution on nonshared memory machines. In large part these conflicting goals were met by having the programmer give concise specifications of aspects of the computation critical to performance, such as the data distribution. Section 2.1 below describes the syntax and semantics of the Kali language. The compiler takes this high-level source code and translates it into detailed message-passing code based on the SPMD (*S*ingle *P*rogram *M*ultiple *D*ata) model [Kar87]. The target code is a single program which can be executed on every processor of the target machine. Section 2.2 briefly sketches an implementation of Kali based on this organization. Chapters 4 and 5 will expand on details of this implementation not covered in depth in Section 2.2. Kali or Kali-like pseudocode will also be used for all programming examples in later chapters.

2.1 Kali Syntax and Semantics

Kali grew from a set of extensions applicable to sequential imperative languages. The syntax described here applies those extensions to BLAZE [MV87], a large-grain dataflow language for expressing scientific computations. (This explains the syntactic differences from [MV89], which applies the same ideas to FORTRAN.) One goal of BLAZE was to make the programming environment as much like traditional sequential environments as possible. Because of this, the sequential features of Kali are quite standard. Section 2.1.1 briefly describes these features. The parallel programming constructs, however, differ somewhat from other languages. Sections 2.1.2 through 2.1.4 describe Kali's three major parallel constructs: processor arrays, data distribution patterns, and the **forall** statement.

Kali also has several constructs for parallelism in addition to the **forall**. Reduction operators relax the restrictions on inter-iteration data dependences to allow summations, multiplications, maximums, and minimums to be calculated across the iterations of **forall**s. Parallel subroutines allow distributed data structures to be manipulated in parallel. Because we do not use these in the main part of this work, we will not discuss them in depth. They do, however, form an important part of a complete language for parallel programming. The reader is referred to [MV89] for details on these constructs.

[1] The name Kali comes from an 8-armed Hindu goddess. In our context, the multiple arms represent the parallelism that we hope to exploit. For more details on the original goddess, see [Stu77].

```
processors Procs : array [ 1..P ] with P in 1..max_procs;

processors Procs2 : array [ 1..P, 1..P ] with P in 1..sqrt_max_procs;
```

Figure 2.1: Kali processor arrays

2.1.1 Sequential Constructs

Outside of explicitly parallel constructs, Kali code is executed sequentially. For these situations, Kali provides the usual complement of imperative constructs. This includes assignment statements, function calls, **for** and **while** loops for iteration, and **if** and **case** statements for conditionals. Because their syntax and semantics so closely resemble those of sequential languages, we will not discuss these constructs in detail. Instead, we emphasize the relationship of these features to the parallel semantics of Kali.

Besides the distributed arrays described in Section 2.1.3, Kali provides scalar variables and undistributed arrays. Pascal-like declarations are used for variables of these types. Semantically, there is a single copy of each of these variables which all processors can access. As will be seen in Section 2.2.1, the actual implementation is somewhat more complex than this, but this need not concern the programmer. Treating scalars as single quantities available to all processors provides a simple, high-level model of the program.

The effects of each sequential statement are felt globally. In particular, assignments to variables have the same effect for all processors regardless of whether the variable is distributed or not. Since all processors have a coherent view of the data, conditions evaluate identically on all processors, avoiding deadlock and race conditions. Distributed arrays may be used in any expression in sequential parts of the code. It is the compiler's responsibility to generate the necessary message-passing code in this situation.

2.1.2 Processor Arrays

The first thing that a Kali program must specify is the array of processors to run the program, which we call the "processor array." The first line in Figure 2.1 declares *Procs* to be a one-dimensional array of processors. Multi-dimensional processor arrays can also be declared, as shown by the declaration of *Procs2* on the second line. (At present, however, Kali does not allow two **processors** declarations in the same scope.) The *Procs* declaration allocates P processors to *Procs*, where P is an integer constant between 1 and *max_procs* chosen when the program is loaded. Our current implementation chooses the largest feasible P; future implementations might use fewer processors to improve granularity or for other reasons. Once chosen, P remains fixed for the duration of the program. It is assumed that each element of the processor array corresponds to one physical processor. Similar comments apply to the *Procs2* declaration. This does two things: it provides a way of naming processors (by indexing into the processor array) and it parameterizes the code by the number of processors.

The ability to name individual processors is needed in order to define data distributions and pass messages. We chose the idiom of processor arrays in part because subscripting provided a familiar naming scheme. We will see in Chapter 3 that it also provides a convenient way to formally define data distribution. Aside from these conveniences, however, there is no magic in describing the set of processors as an array. In particular, the order of array elements does not imply anything about interprocessor connections. It is assumed that the underlying machine can efficiently support virtual arrays of processors, and provide communications between arbitrary processor pairs. This can be accomplished by incorporating routing algorithms in the message-passing routines. The cost of this routing will vary according to topology of the machine and the processor array. Different mappings of physical processors to processor arrays are possible, and this may also affect the communication costs on some machines. We will return to this point in Section 2.2.3.

```
processors Pr : array [ 1..P ] with P in 1..max_procs;

var A : array[ 1..N ] of real dist by [ block ] on Pr;
    B : array[ 1..N ] of real dist by [ cyclic ] on Pr;
    C : array[ 1..N ] of real dist by [ cyclic( 2 ) ] on Pr;
    D : array[ 1..M, 1..M ] of real dist by [ cyclic, * ] on Pr;
    E : array[ 1..M, 1..M ] of real dist by [ *, block ] on Pr;
```

Figure 2.2: Kali distribution patterns in one dimension

Parameterization of a program by the number of processors is important for a number of reasons. It makes the program portable to machines with different numbers of processors. Scaling the program up for larger machines is transparent (up to the upper bound in the **processors** declaration). Similarly, it avoids deadlock if fewer processors are available than expected. These effects can also be achieved in other ways, such as by declaring virtual processors. In a virtual processor environment, the programmer may declare many more processors than actually exist (often one processor for every array element) and it is the compiler's responsibility to map these virtual processors onto the actual processors. We prefer our style of parameterization because it allows full user control of the computation.

2.1.3 Distribution Patterns

As discussed in Section 1.1, an important task in programming nonshared memory machines is data distribution. Kali supports the distribution of arrays, since they are the largest data structure in most Kali programs. All arrays in Kali are declared to be their "natural" size, that is, a single declaration stating the total number of elements on all processors is used. For distributed arrays, no processor stores all of these elements; instead, the compiler generates code so that each processor calculates the section of the array that it stores. This calculation is controlled by the **dist** clause, which provides notations for the most common distribution patterns and allows users to define their own patterns. Kali data distributions can be described by dividing an array into pieces and assigning each piece to a processor. In this section we will use this model to give an intuitive description of distribution patterns. Section 3.2 will define data distributions more formally.

In Kali the number of data array dimensions that can be distributed by predefined patterns is determined by the number of dimensions of the processor array. That is, a one-dimensional processor array like the *Procs* array in Figure 2.1 allows one dimension of each array to be distributed; other dimensions are not divided. We will see later that user-defined distributions need not reflect this. Kali defines three common patterns for dividing an array dimension among processors.

1. **Block** distribution, which divides the indices in the distributed dimension into equally-sized contiguous blocks.

2. **Cyclic** distribution, which allocates indices one at a time to processors, wrapping around when necessary.

3. **Block-cyclic** distribution, which divides the indices into contiguous blocks and allocates these blocks in a cyclic fashion.

The arrays declared in Figure 2.2 demonstrate these patterns on a one-dimensional processor array.

The most common distribution pattern for one-dimensional arrays is block distribution. Array A in Figure 2.2 shows how this is declared in Kali. This pattern groups the array elements into contiguous non-overlapping subsets, storing each subset on a single processor. For example, if $N = 1000$ and $P = 10$, then processor $Pr[1]$ would store elements $A[1]$ through $A[100]$; $Pr[2]$ would store $A[101]$ through $A[200]$; and so on. Figure 2.3 illustrates this for $N = 16$ and $P = 4$. If P does

Figure 2.3: Block distribution in one dimension (array A)

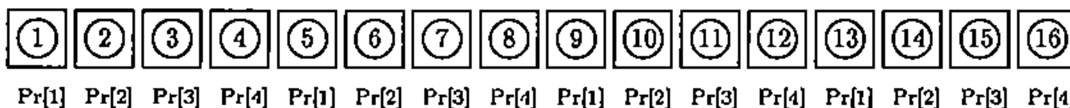


Figure 2.4: Cyclic distribution in one dimension (array B)

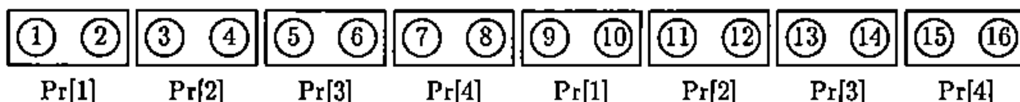


Figure 2.5: Block-cyclic distribution in one dimension (array C)

not divide N, Kali leaves the last (i.e. highest-numbered) processor with a smaller partition than the other processors. Block distribution tends to reduce communication if many references are made to neighboring elements.

If there are P processors, cyclic distribution stores every Pth element on the same processor. The Kali notation for this is shown in the declaration of array B in Figure 2.2. For example, if $N = 1000$ and $P = 10$, then processor $Pr[1]$ would store elements $B[1]$, $B[11]$, $B[21]$, and so on; $Pr[2]$ would store $B[2]$, $B[12]$, $B[22]$, etc. Figure 2.4 illustrates this for $N = 16$ and $P = 4$. Notice that each processor appears several times. This pattern is often useful if only a subrange of the original array will be used. Cyclic distribution then distributes the workload relatively evenly, while block would leave some processors idle.

The above patterns can be combined using a block-cyclic scheme. Kali represents this pattern as a variant of the cyclic distribution, as shown in the declaration of array C in Figure 2.2. This pattern uses a parameter K, which appears in the parenthesis after the keyword **cyclic**. The array is divided into contiguous blocks of size K which are then distributed cyclically among the processors. For example, if $N = 1000$, $P = 10$, and $K = 50$ then processor 1 would store elements $C[1]$ through $C[50]$ and $C[501]$ through $C[550]$. Other processors would have similar sets of elements. Figure 2.5 illustrates this pattern for $N = 16$, $P = 4$, and $K = 2$, as in the actual declaration of C. Notice that each processor appears twice. This pattern is a compromise between **block** and **cyclic** patterns, and is often used when some considerations (such as load balancing) favor **cyclic** distribution and some (such as enhancing data locality) favor **block** distribution. In fact, **block** and **cyclic** distributions can be considered as special cases of **block-cyclic** distribution. When $K = 1$, the distribution is simple **cyclic** distribution; **block** distribution occurs when $K = N/P$.

When the processor array has only one dimension, multi-dimensional array distributions are obtained by applying one-dimensional distribution patterns to a single dimension and not distributing any other dimension. Undistributed dimensions in Kali are marked with an asterisk. For example, the rows of a matrix could be cyclically distributed, as in the declaration of array D in Figure 2.2. Alternately, the columns of a matrix could be distributed by **block**, as shown in the declaration of array E. Figures 2.6 and 2.7 show these distribution patterns for $M = 8$ and $P = 4$. In those figures and in all later two-dimensional distributions, array elements are numbered in their standard order; thus, the upper left corner circle of Figure 2.6 represents $D[1,1]$ and the lower right corner is $D[M,M]$. The considerations in choosing among two-dimensional distributions are similar to those for their one-dimensional cousins.

When the processor array has more than one dimension, additional dimensions of the data

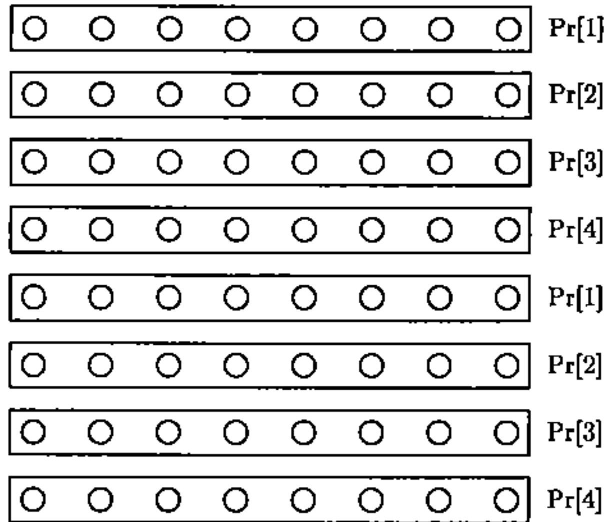


Figure 2.6: Cyclic distribution of rows (array D)

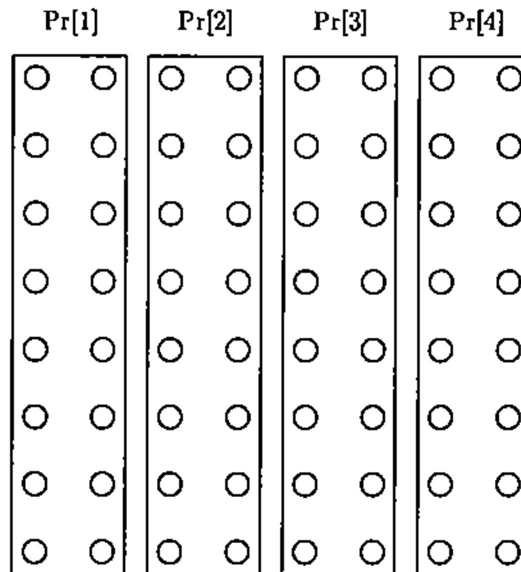


Figure 2.7: Block distribution of columns (array E)

```
processors Procs : array [ 1..P, 1..P ] with P in 1..sqrt_max_procs;

var F : array[ 1..M, 1..M ] of real dist by [ block, block ] on Procs;
    G : array[ 1..M, 1..M ] of real dist by [ cyclic, block ] on Procs;
```

Figure 2.8: Kali distribution patterns in two dimensions

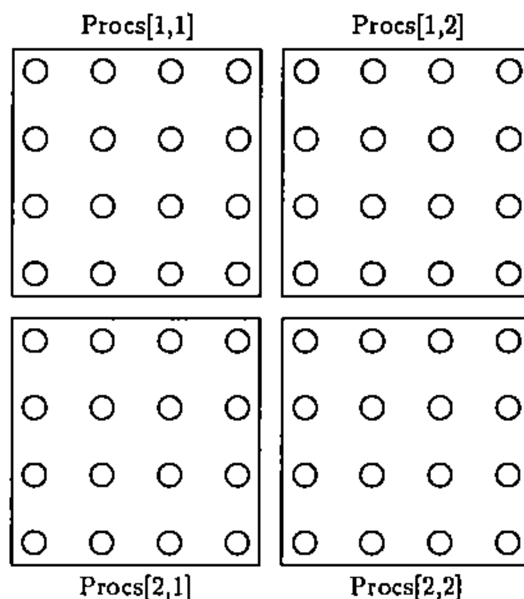


Figure 2.9: Two-dimensional block distribution (array F)

arrays must be distributed. These distributions are formed by taking the tensor products of one-dimensional distribution patterns. In effect, each dimension of the data array is distributed across one dimension of the processor array. If the data array has more dimensions than the processor array, some dimensions must remain undistributed as before. If the data array has fewer dimensions than the processor array, it currently cannot be distributed. In the future, we may relax this restriction to allow low-dimension arrays to be replicated across some dimensions of a multi-dimensional processor array. Figure 2.8 shows two data distributions using a two-dimensional processor array. Array F uses a **block** distribution in both dimensions; this is illustrated in Figure 2.9 for $M = 8$ and $P = 2$. Array G's distribution combines **cyclic** and **block** distribution patterns, as illustrated in Figure 2.10. The two-dimensional **block** distribution is commonly used for solving partial differential equations on a regular domain because it induces relatively little communication. Array G's distribution would be useful for balancing computational loads in some algorithms.

Finally, Kali allows user-defined data distributions. Figure 2.11 shows an example of such a distribution. These distributions must specify two expressions for each possible array index. In the example, the $[i, j]$ term is a dummy parameter that can take on any legal subscript value. The number of variables defined in the term must be the same as the number of dimensions in the array. The **on** clause gives the processor storing $H[i, j]$, in this case $Procs[(i - j) \bmod P + 1]$; there can be only one storing processor for each array element. Figure 2.12 illustrates the distribution generated for array H. It is identical to the skewed distribution described in Section 1.2. This distribution is useful when antidiagonal strips (for example, the dotted box in Figure 2.12) can be computed in parallel. The other expression in the declaration (here, $[(i * N + j - N - 1)/P]$) specifies the storage offset of element $H[i, j]$ on its processor. Section 2.2.4 discusses in some detail how this is used;

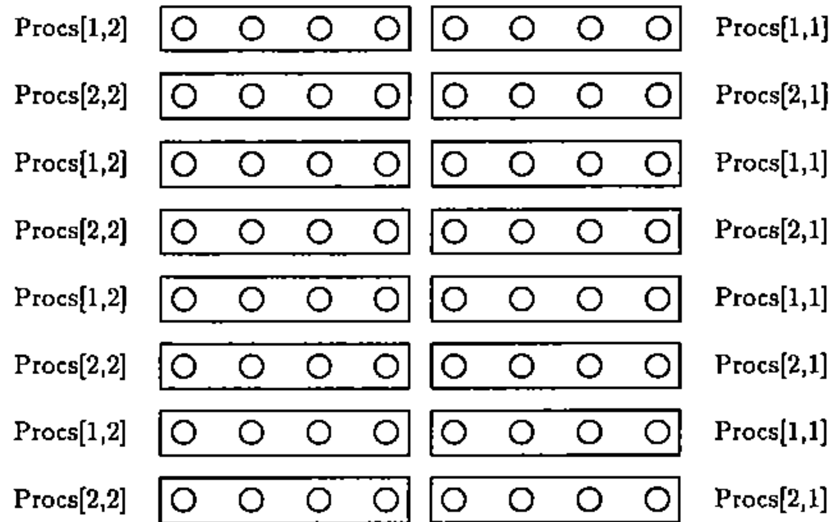


Figure 2.10: Combining cyclic and block distributions (array G)

```
processors Procs : array [ 1..P ] with P in 1..max_procs;

var H : array[ 1..M, 1..M ] of real
        dist by [i,j] => [ (i*N+j-N-1)/P ] on Procs[ (i-j)%P+1 ];
```

Figure 2.11: User-defined data distribution in Kali

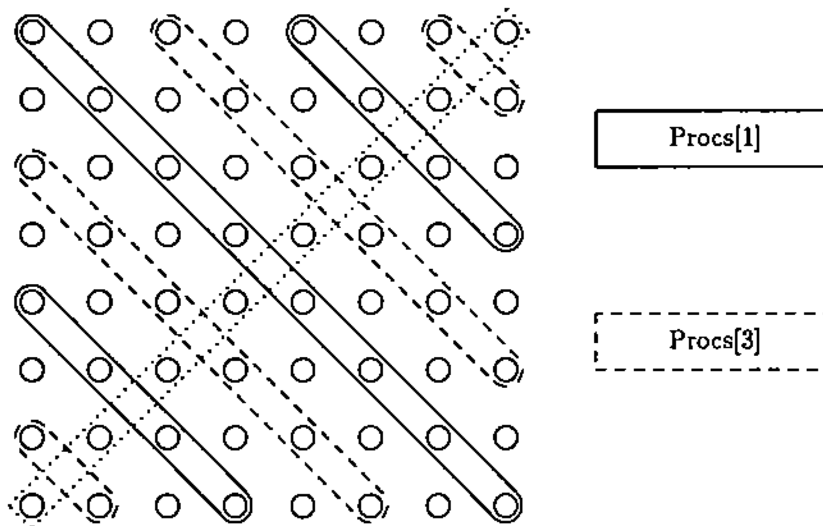


Figure 2.12: Skewed distribution

```
processors Procs : array [ 1..P ] with P in 1..max_procs;

var A : array[ 1..N ] of real dist by [ block ] on Procs;
   B : array[ 1..N ] of real dist by [ cyclic ] on Procs;

forall i1 in 1..N on A[i1].loc do
   A[i1] := B[i1];
end;

forall i2 in 1..N by 2 on A[i2].loc do
   A[i2] := A[i2-1];
end;

forall i3 in 1..P on B[N+1-i3].loc do
   B[i3] := B[i3] + B[N+1-i3];
end;

forall i4 in 1..N/2 on Procs[i4%P+1]
   var temp : real;
do
   temp := B[i4];
   B[i4] := B[N+1-i4];
   B[N+1-i4] := temp;
end;
```

Figure 2.13: Kali **forall** statements

for now, the expression can be thought of as a subscripting formula. Specifying this offset gives up some notational elegance to gain compiler efficiency; future implementations of Kali may make it optional or remove it in favor of other implementation techniques.

2.1.4 Forall Statement

Parallel operations in Kali are specified by **forall** statements, several examples of which are shown in Figure 2.13. The Kali **forall** is semantically identical to the forall described in Section 1.2; that is, it has no inter-iteration data dependences, and thus can have its iterations executed concurrently. Index variables are implicitly declared in the loop header. The range of the index variable is given by the in clause. The first loop, for example, has the same range as a Pascal for loop with the header

> for i1 := 1 to N do

Non-unit strides are also allowed in **forall** ranges, as shown in the second example. Multi-dimensional ranges are also allowed using the syntax of Figure 1.2; in this case, the **forall** iterates over the cross product of the individual ranges. Sequential **for** loops in Kali use the same notation for their ranges. **Foralls** can also have local variables; the syntax for declaring them is shown in the last example. Semantically, the effect of a **forall** is that all iterations are executed in parallel and all processes perform a barrier synchronization until all iterations are complete. The synchronization avoids the possibility of race conditions or deadlock in the surrounding sequential code. Kali **foralls** have two attributes not found in other languages for nonshared memory machines:

1. Each processor executes a subset of the iterations specified by the **on** clause in the **forall** header.
2. The body of the **forall** has no explicit communication statements, even when processors may need to receive data from other processors.

The distribution of **forall** iterations among processors is specified by the **forall**'s **on** clause. In the first **forall** of Figure 2.13, iteration i will be executed on the processor which stores element $A[i]$. In general, the **loc** expression refers to the processor which stores the named array element. Many loops, like the first two examples, use very simple versions of this clause because they are conceptually iterating over elements of an array. Arbitrary expressions can be used in the **loc** expression's subscript, as shown by the third example. The **on** clause can also refer directly to the processor array, if that is more convenient. The last **forall** in the figure shows an example of this. In all cases, a processor executes all **forall** iterations which refer to it when their **on** clauses are evaluated. Section 2.2.5 outlines techniques which can avoid explicitly checking the **on** clauses of every iteration. At present, the **on** clause is mandatory. As mentioned in Section 1.3, heuristics for choosing the distribution of computations are possible; one of these will be incorporated in a later version of Kali, making the **on** clause optional.

The rationale for avoiding explicit communication statements was explained in Section 1.2. Message-passing programs must be written at too low a level. Hiding the communication raises the level of programming substantially, since details of message sources, destinations, and contents need not directly concern the programmer. In Kali, communication is implicitly generated when a processor references nonlocal data in a **forall**. It is the compiler's responsibility to recognize this situation and generate code to perform the low-level message passing. Chapters 3, 4, and 5 are devoted to solving exactly this problem.

2.2 Kali Implementation

In this section we describe the implementation of Kali on nonshared memory machines. After an overview of the basic ideas in Section 2.2.1, each subsection here sketches the implementation of the features defined in the corresponding subsection of Section 2.1. These descriptions are not meant to be comprehensive, particularly in the case of Section 2.2.5. The details of implementing **foralls** occupy Chapters 3, 4, and 5.

2.2.1 Basic Concepts

We have chosen to implement Kali programs by generating code based on the SPMD (*S*ingle *P*rogram, *M*ultiple *D*ata) model of computation [Kar87]. In this model, all processors execute copies of the same program code, parameterized by the processor id. The processors execute asynchronously, however. This is similar to the form of Kali source programs, in which all processors execute the same **forall** statement, but may execute different statements within the **forall** because of conditionals. SPMD programming should not be confused with SIMD computation, in which all processors execute synchronously. In the SPMD model, processors may follow different branches of the code, even to the point of executing completely different functions. This is not possible in SIMD models. In the Kali implementation, only a limited amount of asynchrony is used to allow processors to calculate their own loop bounds, perform different numbers of **forall** iterations, take different branches of conditionals, and build dynamic data structures. The original Kali code, however, may contain additional sources of asynchrony, such as calling complex functions based on the number of the **forall** iteration.

One consequence of the SPMD model is that code outside of **forall** statements is duplicated on all processors. There are two ways this can be implemented.

1. By duplicating sequential computations on all processors

2. By using a conditional to make one processor the master and all others slaves. The master then performs all sequential computations and sends the results to the slaves.

Our implementation duplicates sequential computations on all processors. This decision was based on both the method's ease of implementation and on expected performance gains in a nonshared memory

environment. Section 2.2.2 describes some modifications to this model required by computations on distributed data. Note that duplicating sequential computations automatically duplicates the control flow, because it duplicates any conditional branching. This implies that all processors must have a consistent view of the state of the program to avoid deadlock.

Because Kali has no explicit communication constructs, the compiler must insert them into the compiled code. Messages must be generated in two situations: in parallel code when one processor may reference data on another, and in sequential code when any distributed data is referenced. The reasons for message-passing in the first situation are clear. The second situation requires communication because the code is executed on all processors, but the distributed data is stored on only one. Because all processors execute the same program, message-passing statements must usually be guarded by conditionals to ensure that only the appropriate processors participate in the communication. All these considerations add complexity to the compiler; that is the price of providing a high-level language to the user.

One can think of the process of generating the communication instructions as a source-to-source translation from Kali into a message-passing language. This is, in fact, how the Kali compiler works; the target language is C code with subroutines for sending and receiving messages. Because the generated code tends to be opaque, we will not present it directly. Instead, we will illustrate translations with a fairly detailed pseudocode based on Kali. The major changes made in this pseudocode are the addition of message-passing statements and the assumption that the code is being executed in SPMD mode. As is traditional with pseudocode, complex data structures will not be shown in detail.

2.2.2 Sequential Constructs

Undistributed variables in Kali (i.e. scalars and undistributed arrays) are duplicated on all processors. This allows these variables to be used by any processor at any point in the code without stopping for communication. Because Kali's semantics specify that all processors have identical views of the data, the values of these variables must be kept consistent. This is achieved by careful generation of the code for assignment statements, as explained below.

The left-hand side of an assignment statement can refer to either a distributed array element or an undistributed variable. Similarly, the computation on the right-hand side may require data from distributed arrays or it may not. The combinations of these possibilities give four classes of assignment statements in sequential code, all of which need different implementations.

Figure 2.14 shows the cases of assignment to undistributed variables. Statement 1 shows the simplest possible case of assignment: no distributed array elements are present. In this case, all processors have consistent copies of all of the data needed to compute the right-hand side. All processors can therefore perform the same calculation, obtaining the same result, and assign that result to their copy of the left-hand side. Note that this preserves the consistent state of the copies of the variable X. Statement 2 shows the assignment of a distributed array element to an undistributed variable. Since all processors need data which is stored on only one processor, a broadcast is used to give all processors a copy. The test $A[I] \in \mathit{local}(p)$ checks whether $\mathit{Proc}[p]$ stores $A[I]$. This notation is defined more formally in Section 3.2, and the implementation of the test is described in Section 2.2.4. For now, it is enough to say that $\mathit{local}(p)$ is the set of array elements stored on $\mathit{Procs}[p]$; therefore, the test is true on a processor if that processor must broadcast the value, and false if it must receive the broadcast. The expression $\mathit{Procs}[*]$ in communication statements denotes a broadcast, either sent to all processors or received from an anonymous processor. Single-destination messages identify the correct sender and receiver. If there are several references to distributed variables, an **if** statement and broadcast are required for each reference. Once all processors have received all of the data, they all compute the right-hand side and make the assignment. Once again, consistency is guaranteed since all processors perform the same computation on the same data.

Figure 2.15 shows two assignments to distributed array elements. Statement 3 shows the assignment of an undistributed value to a distributed array. In this case, all processors simply check whether they own the left-hand side of the assignment, and perform the calculation if they do.

```
processors Procs : array [ 1..P ] with P in 1..max_procs;

var I : integer;
   X : real;
   A : array[ 1..N ] of real dist by block on Procs;

        Kali code                    Message-passing code for Procs[p]
   -- Statement 1                   -- Statement 1
   X := X * 2;                      X := X * 2;

   -- Statement 2                   -- Statement 2
   X := A[I];                       if ( A[I] ∈ local(p) ) then
                                       tmp := A[I];
                                       send( tmp, Procs[*] );
                                    else
                                       tmp := recv( Procs[*] );
                                    end;
                                    X := tmp;
```

Figure 2.14: Implementing assignments to undistributed variables

```
processors Procs : array [ 1..P ] with P in 1..max_procs;

var I : integer;
   X : real;
   A : array[ 1..N ] of real dist by block on Procs;

        Kali code                    Message-passing code for Procs[p]
   -- Statement 3                   -- Statement 3
   A[I] := X;                       if ( A[I] ∈ local(p) ) then
                                       A[I] := X;
                                    end;

   -- Statement 4                   -- Statement 4
   A[I] := A[I*2];                  if ( A[I] ∈ local(p) ) then
                                       if ( A[2I] ∈ local(p) ) then
                                             tmp := A[2*I];
                                       else
                                             q := processor storing A[2*I]
                                             tmp := recv( Procs[q] );
                                       end;
                                       A[I] := tmp;
                                    else
                                       if ( A[2I] ∈ local(p) ) then
                                             q := processor storing A[I]
                                             send( A[2*I], Procs[q] );
                                       end;
                                    end;
```

Figure 2.15: Implementing assignments to distributed variables

```
processors Procs : array [ 1..P ] with P in 1..max_procs;

var I : integer;
   A : array[ 1..N ] of real dist by block on Procs;
   Permute : array[ 1..N ] of integer dist by block on Procs;
```

Kali code	Message-passing code for *Procs[p]*
A[I] := A[Permute[I]];	see below

```
if ( Permute[I] ∈ local(p) ) then
   tmp1 := Permute[I];
   send( tmp1, Procs[*] );
else
   tmp1 := recv( Procs[*] );
end;
if ( A[I] ∈ local(p) ) then
   if ( A[tmp1] ∈ local(p) ) then
         tmp2 := A[tmp1];
   else
         q := processor storing A[tmp1]
         tmp2 := recv( Procs[q] );
   end;
   A[I] := tmp2;
else
   if ( A[tmp1] ∈ local(p) ) then
         q := processor storing A[I]
         send( A[tmp1], Procs[q] );
   end;
end;
```

Figure 2.16: Implementing assignment with indirection

Note that processors that do not own they left-hand side need not even compute the right-hand side. Statement 4 is the most complex case: a distributed array element being assigned to another distributed array element. In this case, the processor owning the left-hand side collects all of the data needed to compute the right-hand side, either from its own sections of distributed arrays or from messages received, performs the computation, and makes the assignment. Other processors check whether they have data needed for the right-hand side, and send it to the processor doing the computation if they do. The inner **if** statements must be duplicated for each reference to a distributed array on the right-hand side, but the outermost **if** statement will not change. The communication statements and their associated conditions can be avoided if the compiler can detect that a reference on the right-hand side is stored on the same processor as the left-hand side. This occurs, for example, when the left-hand and right-hand references are the same, as in an increment of the array element.

The preceding discussion assumed that all subscripts were computable on all processors in order to perform the locality checks. This will not be the case if a distributed array is used as an index vector. Figure 2.16 illustrates this case and the generated code. The computation must proceed in two phases. First the value of $Permute[I]$ is broadcast to all processors, and then the algorithm of Figure 2.15 can be applied. Additional levels of indirection can be handled recursively, starting with the innermost subscripts.

Expressions in other sequential constructs can be handled by methods similar to those for assignments. Since sequential control flow is duplicated on all processors, loop bounds, while conditions, and expressions in conditionals can be treated as assignments to scalars. Thus, expressions with no distributed array references can be evaluated on all processors immediately, while expressions using

distributed arrays require broadcasts. The consistency arguments for scalar assignment guarantee that all processors will then follow the same branches of the program.

The strategy given in this section generates many small messages, which severely impacts the performance of sequential Kali code that accesses distributed arrays. Much optimization in the area of aggregating messages is possible and sorely needed in the current compiler. That this has not been done yet is a matter of our priorities; we felt it was more important to test our ideas for the parallel features than to optimize the sequential parts. Techniques similar to those of [CK88, RP89, Rog90, ZBG88] are applicable to Kali and should produce substantial improvements in the generated code. We plan to incorporate these optimizations in the near future.

2.2.3 Processor Arrays

To implement the processors declaration on a parallel machine, each processor must determine the number of processors available to the program and its own processor id. These can determined by simple system calls on most commercially available machines. The number of processors is used to set the value of the bound variable in the processors declaration on each processor. Similarly, each processor determines its location in the Kali processor array from the system processor id. At present, we use a naive scheme to convert from system processor ids to processor array indices; we simply add the lower bound of the processor array to the 0-based id. Other mapping schemes are also possible. In particular, Gray codes allow adjacent elements in the processor array to be mapped to physically connected hypercube processors [Gil58]. Changing the Kali compiler to use this mapping would only mean changing a single assignment during program initialization and modifying less than 10 lines in the run-time environment. Multi-dimensional processor arrays are not currently implemented, but the principle of computing the processor array index would be similar. A mapping, such as row-major ordering, would be defined from the one-dimensional processor id onto the higher-dimension processor array index set, and the index computed from this mapping.

2.2.4 Distribution Patterns

The effect of a dist clause in an array declaration is fourfold.

1. A formula is generated to compute the processor storing each array element.
2. Each processor generates a representation of the set of array elements it stores. This representation need not be explicit.
3. The subscripting formula for the array is generated. This formula translates subscripts based on the global array into an offset into the local section of the array.
4. Space for the local section of the array is allocated on each processor.

For predefined distributions such as block and cyclic, these computations can be done simply; user-defined distributions require more complex machinery. More precisely, the information for predefined distributions can be calculated in the compiler, while the calculations for user-defined distributions must be done at run-time. We will examine each of these facets of distributed arrays. To make our discussion more concrete, we will refer to the arrays defined in Figure 2.17. All arrays there are defined to be 0-based in order to make the expressions in this section simpler. The compiler derives similar expressions for arrays with other lower bounds.

We first consider how the processor storing an array element can be computed. In addition to being commonly used, the predefined distribution patterns have particularly simple expressions for finding that processor. Let the processor storing array element $A[i]$ be $Procs[proc_A(i)]$, and similarly for the other arrays. Then

$$proc_A(i) = \left\lfloor \frac{i}{\lceil N/P \rceil} \right\rfloor \tag{2.1}$$

```
processors Procs : array [ 0..P-1 ] with P in 1..max_procs;

var A : array[ 0..N-1 ] of real dist by [ block ] on Procs;
   B : array[ 0..N-1 ] of real dist by [ cyclic ] on Procs;
   C : array[ 0..N-1 ] of real dist by [ cyclic(K) ] on Procs;
   D : array[ 0..N-1, 0..N-1 ] of real
         dist by [i,j] => [ (i*N+j)/P ] on Procs[ (i-j)%P ];
```

Figure 2.17: Sample distribution patterns

$$proc_B(i) = i \bmod P \tag{2.2}$$

$$proc_C(i) = \left\lfloor \frac{i}{K} \right\rfloor \bmod P \tag{2.3}$$

The ceiling function in Equation 2.1 can be eliminated if N is divisible by P. Expressions for arrays with predefined patterns and undistributed dimensions can be obtained by simply ignoring those dimensions. The *proc* function for user-defined distributions is given by the **on** clause. In this case,

$$proc_D(i,j) = (i-j) \bmod P \tag{2.4}$$

The Kali compiler defines these functions as macros and inserts them where needed. All locality checks, for example, are performed by comparing this macro to the processor array index.

A processor's local subset of array elements is used primarily in distributing **forall** iterations. With this in mind, we define descriptions of those sets which allow easy enumeration. Foreshadowing the definitions of Chapter 3, we will denote the set of local elements of array A on processor p as $local_A(p)$. For the **block** distribution, the lower and upper bounds of a processor's block describe the set completely. We can thus define

$$local_A(p) = \left\{ \left\lceil \frac{N}{P} \right\rceil \cdot p, \ldots, \left\lceil \frac{N}{P} \right\rceil \cdot (p+1) - 1 \right\} \tag{2.5}$$

Similarly, $local(p)$ for the cyclic distribution can be described by the processor p's local lower bound, the array upper bound, and number of processors.

$$local_B(p) = \left\{ p, p+P, p+2P, \ldots, p + \left\lfloor \frac{N}{P} \right\rfloor P \right\} \tag{2.6}$$

Block-cyclic distributions require a lower bound, upper bound, and the block size to describe the $local(p)$ functions. For iteration purposes, this is implemented as a pair of perfectly nested loops, the outer running over the possible blocks (using non-unit strides) and the inner single-stepping through a block. Section 3.2 will give an explicit expression for $local_C(p)$. For user-defined distributions, determining $local(p)$ is more complex. Essentially, what must be done is to create a list of all local array elements on the local processor. In the current implementation, this is done by looping over all array indices and checking the generated values of the **on** clause. This need only be done once for each distinct **dist** clause, but even so the startup overhead is significant. Later **foralls** can then iterate through the list in linear time.

For both predefined and user-defined distributions, it is necessary for the compiler to generate indexing calculations into the local section of the array from subscripts based on the global array. This global-to-local translation is conceptually similar to the familiar row-major ordering and column-major ordering used for two-dimensional arrays; it translates the programmer's coordinate system (indices into the global array) into machine-usable addresses (offsets into the local section). To optimize storage usage, the subscripting formula should compute, for every index, the number of array elements in the same local section with smaller indices. It is also advantageous to use the

same formula on all processors. The Kali compiler generates these formulas as macros called *sub* functions. For predefined distribution patterns, analytic formulas for these functions are available.

$$sub_A(i) = i - \left\lceil \frac{N}{P} \right\rceil \cdot p \tag{2.7}$$

$$sub_B(i) = \left\lfloor \frac{i}{P} \right\rfloor \tag{2.8}$$

$$sub_C(i) = \left\lfloor \frac{i}{KP} \right\rfloor \cdot K + i \bmod K \tag{2.9}$$

In practice, the divisions by P can often be replaced by shift operations, since P is almost always a power of two and the numerators are nonnegative. If there are undistributed dimensions, then Kali treats the value of the sub function as the first subscript of a new array, followed by the subscripts of undistributed dimensions. The new array then uses row-major ordering to produce the final offset. For user-defined distributions, the first expression in the **dist** clause gives the addressing formula. For the example,

$$sub_D(i) = \left\lfloor \frac{i \cdot N + j}{P} \right\rfloor \tag{2.10}$$

(The floor functions are the result of integer division truncation.) Each processor p also calculates the lowest value of this formula while computing $local(p)$ and subtracts it during subscript calculations to convert to 0-based addressing.

Allocating space for the array is straightforward. Since the predefined patterns divide the array evenly among processors, each processor allocates space for $\lceil N/P \rceil$ elements for the first three arrays. (Some adjustment must be made for block-cyclic distributions, which may generate unbalanced loads.) For user-defined distributions, the number of elements on processor p is calculated during the computation of $local(p)$, and this is used to manage storage.

2.2.5 Forall Statement

There are two major issues in implementing forall statements. The first is distributing the **forall** iterations, and the second is generating communication statements. Chapters 4 and 5 give detailed explanations of how the communication is implemented, so we mention only the key issues here. Distributing the iterations is a simpler problem, which we address more fully in this section.

The implementation of a **forall** requires each processor p to execute only the iterations specified for it in the **on** clause. There are several possible cases of this, as illustrated in Figure 2.18. The figure denotes the set of iterations executed on processor p as $exec(p)$, which is the notation used in Chapter 3. The expressions for $exec(p)$ given there are mathematical descriptions for the sets; in this paragraph we describe how they may be implemented. In the first **forall** (h) of Figure 2.18, processor p must simply iterate over the elements of array A stored locally. This can be done using the appropriate representation of the $local(p)$ set described earlier. Similar tactics can be used for the second **forall** (i) after adjusting the bounds to conform with the **forall** index range. If A were distributed by a user function, this would involve iterating through the $local(p)$ list and checking whether the list values were in the correct range. The third **forall** (j) requires a method of constructing the inverse of function f. Chapters 4 and 5 show how this can be done; in the worst case, these require each processor to form a list of its iterations. The fourth **forall** (k) is the simplest loop; iteration k is executed on processor k. Finally, the fifth **forall** (l) is a generalization of the previous one which again requires a function inverse to be found.

As we have stated before, it is the compiler's responsibility to generate explicit message-passing code where it is needed. This is particularly true of **forall** statements. Because all processors are not following the same thread of control within a **forall**, the strategy of Section 2.2.2 for generating communications will not work. **Foralls** do have a compensating advantage, however: the data needed during the **forall** is available before execution of the **forall** begins. This is a consequence

```
processors Procs : array[ 0..P−1 ] with P in 1..max_procs;
var A : array[ 0..N−1 ] of real dist by [ block ] on Procs;

forall h in 0..N−1 on A[ h ].loc do

forall i in low..high on A[ i ].loc do

forall j in 0..N−1 on A[ f(j) ].loc do

forall k in 0..P on Procs[ k ] do

forall l in 0..P−1 on Procs[ g(l) ] do
```

(a) Kali forall headers i, j, k, and l

$$
\begin{aligned}
exec_h(p) &= local_A(p) \\
exec_i(p) &= \{low, low+1, \ldots, high\} \cap local_A(p) \\
exec_j(p) &= \{1, 2, \ldots, N\} \cap f^{-1}(local_A(p)) \\
exec_k(p) &= \{p\} \\
exec_l(p) &= g^{-1}(p)
\end{aligned}
$$

(b) Corresponding $exec(p)$ sets

Figure 2.18: Possible cases for forall statements

of the lack of data dependences between forall iterations. Chapters 3, 4, and 5 show that it is also possible, both in theory and in practice, to predict what data a processor will need for its **forall** iterations. Similarly, if a processor modifies data stored on another, the update can be deferred until all iterations are completed. Elements assigned in this way can also be identified. Thus, a processor could implement a **forall** with the following strategy:

- Identify all nonlocal data accessed on this processor and all local data that other processors access.
- Exchange data to be used during forall iterations with other processors.
- Execute the iterations on this processor.
- Exchange data that was modified during the **forall** with other processors.

The exchanges only involve data used or modified on a processor other than its home. This is nearly the strategy that the Kali compiler actually uses. The actual code generated appears in Section 3.1. The idea of computing all data used in a forall and sending it at the start of the **forall** remains in the final form of the code. This is the basic idea behind our methods: to derive a description of the data that must be communicated, and to use that description in generating code. The next three chapters show how this concept applies to forall statements.

Communications also play a role in the synchronization at the end of a **forall** statement. The relatively expensive barrier synchronization can be avoided by noting that processors can only interfere with each other by communicating. If a processor does not receive any messages before it completes its **forall** iterations, then no other processor can alter its private memory. Similarly, if it does not send messages until the completion of its iterations, a processor will only send out the correct values. Both of these conditions are met by the Kali implementation. Therefore, no explicit barrier synchronization is done at the end of a forall construct.

Chapter 3

A Model for Data Distribution and Message Generation

The remainder of this thesis concentrates on the generation of message-passing code for **forall** statements. In this chapter we develop a model that will serve as a basis for the code generation. Chapters 4 and 5 will then show how the model can be implemented. The model and derived formulas here are essentially identical to those in [KM89, KMV90, Koe88]. Section 3.1 introduces this model by showing an outline of the target code and discussing the information required to implement it. The next four sections each formalize one piece of that information. Section 3.2 introduces the $local(p)$ function which models data distribution. Similarly, Section 3.3 describes the $exec(p)$ function, a model for distributing the iterations of a parallel loop. Because Kali's syntax specifies these functions directly, we treat them as basic building blocks for the other parts of the model. Section 3.4 defines *send_set* and *recv_set*, the functions which control communication in the generated program, using *local* and *exec*. Section 3.5 gives a similar treatment to *local_iter* and *nonlocal_iter*, functions that control the computation of the **forall**. This completes the main part of the model. Section 3.6 considers some extensions to the basic model, and Section 3.7 introduces two approaches to implementing the model. Chapters 4 and 5 will each examine one of these approaches in detail.

3.1 The Structure of Generated Code

This section describes in general terms the structure of the code generated for a single **forall** statement. Although we give no formal definitions here, we will use the same notation developed in the next four sections and give informal definitions. This will allow the reader to refer back to the descriptions here.

For concreteness, we will base our discussion on the model program given in Figure 3.1, which has some significant simplifying assumptions. In particular, it is assumed that:

1. $A[f(i)]$ is the only array reference in the loop that can induce communication.
2. $A[f(i)]$ is an r-value rather than an l-value, that is, it is read rather than written.
3. $A[f(i)]$ is always accessed in the loop, that is, there are no conditionals to alter control flow around the reference.

Section 3.6 shows how these assumptions can be relaxed. Other specific features of the figure do not represent assumptions. In particular, the exact **dist** and **on** clauses shown could be any legal clauses. The expressions derived in Sections 3.2 through 3.5 will apply to any distributions of arrays and **forall** iterations.

```
processors Procs : array[ 1..P ] with P in 1..max_procs;

var A, New_A : array[ 1..N ] of real dist by [ block ] on Procs;

forall i in 1..M on New_A[i].loc do
   New_A[i] := A[ f(i) ];
end;
```

Figure 3.1: Example **forall** statement for definition of sets

Code on processor p:

- Generate communication and iteration sets
  - $local(p)$ = Array elements stored on p.
  - $exec(p)$ = Iterations to be performed on p.
  - For all $q \neq p$, $send_set(p, q)$ = Array elements sent from p to q.
  - For all $q \neq p$, $recv_set(p, q)$ = Array elements received by p from q.
  - $local_iter(p)$ = Iterations on p that access only local data.
  - $nonlocal_iter(p)$ = Iterations on p that access some nonlocal data.
- For all q with $send_set(p, q) \neq \phi$, send message containing $send_set(p, q)$ to q.
- Execute computations for iterations in $local_iter(p)$, accessing only local arrays.
- For all q with $recv_set(p, q) \neq \phi$, receive message with $recv_set(p, q)$ from q.
- Execute computations for iterations in $nonlocal_iter(p)$, accessing local arrays and message buffers.

Figure 3.2: Implementing a **forall** on a nonshared memory machine

Each processor must complete three major tasks in order to correctly implement the program of Figure 3.1:

- *Generate information* needed for passing messages and controlling iteration.
- *Exchange data* with other processors so that all processors have the data needed for their computations.
- *Execute computations* using local data and data from received messages.

A more detailed outline of how these tasks are ordered is given in Figure 3.2.

The first step of Figure 3.2 is to generate the information that will be used later. Each processor needs four pieces of information to complete the above tasks.

1. The set of array elements that it stores locally.
2. The set of **forall** iterations that it must execute.
3. The sets of array elements that must be sent and received in messages.
4. Two subsets of the set of iterations: those which access *only* local data and those which access *some* nonlocal data.

The usefulness of the first two sets is obvious. They are also used in computing the remaining information. On processor p they are called $local(p)$ and $exec(p)$. Note that $local(p)$ is determined by the data distribution pattern and $exec(p)$ by the Kali **on** clause. Because of this close relationship with the source program, we will refer to these sets as *basic sets*. The next pair of sets is needed to control the communication with other processors. For every pair of processors p and q there will be sets $send_set(p,q)$ and $recv_set(p,q)$, which have the obvious meanings. We will refer to these sets as the *communication sets*. Finally, the iteration subsets are used to overlap computation and communication, as explained below. The iterations on processor p that need no data from other processors are collected in $local_iter(p)$. Iterations on p that access any data from any other processor make up $nonlocal_iter(p)$ We will refer to these sets as the *iteration sets*.

The next logical task in implementing a **forall** is to perform any necessary communication. This is split into two parts in Figure 3.2; sending the messages is done first, and receiving messages comes later. This splitting will be explained below. Here we concentrate on the communication alone. Since $A[f(i)]$ in Figure 3.1 is an r-value (and this is the only array reference that can cause communication), the only messages needed in the implementation will be for reading nonlocal data. Since there are no inter-iteration dependences in a **forall**, the data for these messages will be available at the beginning of the loop and will not be overwritten within the loop. Thus, the data can be passed as messages at any time before it is needed. Our implementation sends the messages as soon as the data is available, that is, as soon as $send_set(p,q)$ is known. This provides the maximum time for messages to reach their destinations before they are needed. Similarly, messages can be received at any time before their actual use. Our implementation performs all receives in a block immediately before the first nonlocal value is needed. This strategy is called *prefetching* and is quite effective, but it is not the only possible strategy. We will review other possibilities in Chapter 7 in our discussion of related work.

The final logical task in implementing a **forall** is the actual computation. This task is split into two parts and interwoven with the communication task. This organization is used to gain efficiency. If some iterations of the **forall** on processor p use only data stored on p, then those iterations can be executed before any incoming messages have been received. This observation can be exploited to overlap computation and communication by grouping all iterations which use only local data together. These local iterations can be executed without waiting for any messages to be received. The remaining iterations, which depend on data received in messages, must be executed after the messages have been received. Combining this overlap strategy with the prefetching strategy explained above results in the alternation of communication and computation shown in Figure 3.2.

A few final points should be made about Figure 3.2. Each processor only needs to generate its own sets. For example, processor 1 needs no information about $local_iter(2)$ or $send_set(3,6)$. This reduces the amount of information required on each processor. It should also be noted that the sets need not be explicitly generated in all cases; as we will see in Chapter 4, they can often be represented by parameters computed by the compiler. Finally, it should be noted that Figure 3.2 strongly incorporates the simplifying assumption that no nonlocal array elements are used as l-values; the prefetching strategy would not work otherwise. Section 3.6 addresses removing this restriction. We will continue to make this assumption for the next two sections, however. The other assumptions, although they will be used in deriving the formulas of Sections 3.4 and 3.5, do not affect the outline of the generated code.

3.2 Data Distribution

The fundamental task of data distribution is to specify which processors in a nonshared memory machine will store each element of a shared data structure in their private memories. This is done by providing a mapping between the set of processors on a parallel machine and the set of data items to be stored. This mapping is not necessarily one-to-one. Going from data items to processors, there will usually be more data items than processors, so each processor must store more than one datum. In the other direction, it is sometimes advantageous to store several copies of the same data item.

Two particular cases of this are of interest:

1. Scalars and small arrays are usually duplicated across all processors.
2. In practice, it is common to have a small area of "overlap" between the regions stored on neighboring processors to reduce communication.

A general model of data distribution must allow these types of copying.

The first subsection below gives a mathematical model of data distribution for arrays. Subsections 3.2.2, 3.2.3, and 3.2.4 will then give concrete examples of the use of the formalism in Subsection 3.2.1. Included in these examples will be the formulas for Kali distribution patterns presented in Section 2.1.3.

3.2.1 The Mathematical Model

We describe a data distribution by giving the set of array elements stored on each processor. Mathematically, this can be modeled as a function from processors to sets of array elements which we call the *local* function.

Definition 3.1 *Let Procs be the set of processors and Elem the set of elements of an array A. Then*

$$local : Procs \rightarrow 2^{Elem} : local(p) = \{\, a \in Elem \mid a \text{ is stored on } p \,\} \tag{3.1}$$

(Here, 2^S is the class of subsets of set S.) A graphical example of such a *local* function is shown in Figure 3.3. There, the *Procs* set is shown as four small squares, the *Elem* set is shown as a large subdivided square, and the *local* function itself is indicated by the arrows from the *Procs* set to regions of the *Elem* set. Note that in this scheme there is no problem with data copying for overlapped distribution patterns. The only consequence is that the *local* sets of distinct processors are not disjoint. This is the case for processors 0 and 3 in the figure; their overlap is shown as the cross-hatched area. In the examples that follow, we will represent *Procs* and *Elem* by their index sets, which will be tuples of small integers. Also, if there is an ambiguity as to the identity of the array, we will use the array or distribution name as a subscript.

Other approaches to data distribution [CK88, GJG88, RAP87, RP89] have taken a different path toward formalizing the distribution. Generally, these approaches define a function

$$proc : Elem \rightarrow Procs : proc(a) = p, \text{ where } p \text{ is the processor storing } a \tag{3.2}$$

Note that this is the way user-defined distributions are declared in Kali. If every element is stored on exactly one processor, then the two approaches are equivalent. In this case, *local* is simply $proc^{-1}$. If an element can be stored on more than one processor, however, the two methods are not equivalent. It is not obvious how such a distribution scheme could be modeled using a single-valued *proc* function. One possible way around this problem is to redefine *proc* as

$$proc : Elem \rightarrow 2^{Procs}$$

This allows copied elements to be modeled by elements e such that $proc(e)$ has more than one element. In this case, *local* and *proc* can be defined in terms of each other by

$$\begin{aligned} local(p) &= \{\, e \mid p \in proc(e) \,\} \\ proc(e) &= \{\, p \mid e \in local(p) \,\} \end{aligned}$$

Such a redefinition of *proc*, however, loses some of the conceptual clarity claimed by other researchers. Because the *proc* definition is equivalent to the *local* function, we will use the *local* formalism throughout this paper.

The next several sections show the *local* functions for several distribution patterns in wide use. For one-dimensional data and processor arrays, we assume that the data array has N elements

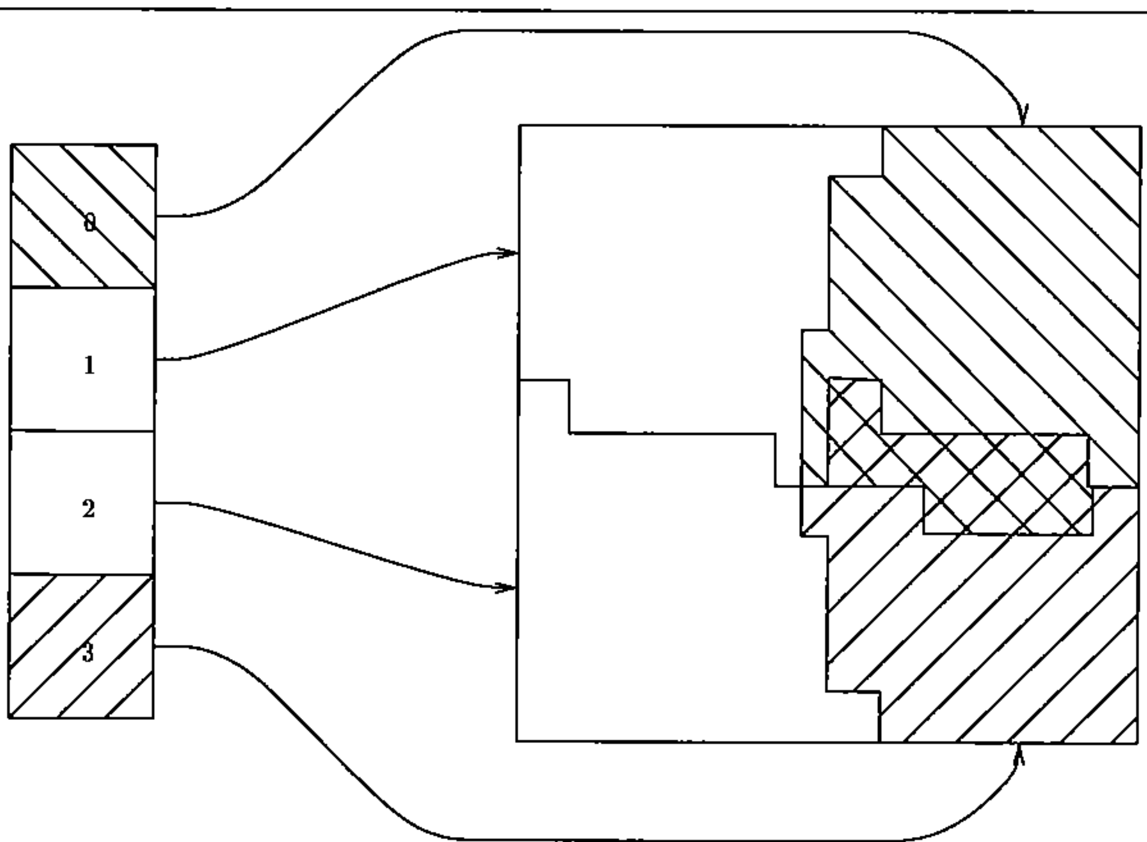


Figure 3.3: Sample *local* function

and there are P processors available. We also use 0-based indexing for both arrays and processors, making the index sets

$$\begin{aligned} Procs &= \{0,1,2,\ldots,P-1\} \\ Elem &= \{0,1,2,\ldots,N-1\} \end{aligned}$$

Similarly, for multiply-dimensioned arrays, we assume the index of the ith dimension runs from 0 to $N_i - 1$ for data arrays and from 0 to $P_i - 1$ for processor arrays. We will also assume that N (or N_i) is divisible by P (P_i) where it simplifies the formulas.

3.2.2 One-Dimensional Distribution Patterns

The most common distribution patterns for one-dimensional arrays are the **block**, **cyclic**, and **block-cyclic** distributions described in Section 2.1.3. Informal definitions of the *local* functions for these patterns were given in Section 2.2.4. We give definitions for these functions without further comment.

$$local_{BLOCK}(p) = \left\{ i \mid \frac{N}{P} \cdot p \leq i < \frac{N}{P} \cdot (p+1) \right\} \tag{3.3}$$

$$local_{CYCLIC}(p) = \{ i \mid i \equiv p \pmod{P} \} \tag{3.4}$$

$$local_{BLOCK-CYCLIC(K)}(p) = \left\{ i \mid \left\lfloor \frac{i}{K} \right\rfloor \equiv p \pmod{P} \right\} \tag{3.5}$$

The K in the definition of $local_{BLOCK-CYCLIC(K)}$ is the block size parameter to that distribution. Equations 3.3 and 3.4 are equivalent to 2.5 and 2.6.

There are two approaches to handling array sizes which are not divisible by the number of processors in **block** distributions. Section 2.2.4 describes Kali's method, which is to leave the last processor with less work than the others. This method has the advantage of simple formulas for its implementation, but can seriously underutilize one processor. Another method is to divide the "excess" elements evenly among the lowest-numbered processors. This method produces the *local* function

$$local(p) = \begin{cases} \{ i \mid (\lfloor \frac{N}{P} \rfloor + 1)\, p \leq i < (\lfloor \frac{N}{P} \rfloor + 1)\,(p+1) \} & \text{if } p < N \,\%\, P \\ \{ i \mid \lfloor \frac{N}{P} \rfloor\, p + N \,\%\, P \leq i < \lfloor \frac{N}{P} \rfloor (p+1) + N \,\%\, P \} & \text{otherwise} \end{cases} \tag{3.6}$$

Note that this distribution guarantees that the number of elements assigned to two processors do not differ by more than 1.

The two types of **block** distribution are special cases of a generalized block scheme. Many distributions assign contiguous sections of arrays to each processor. These can be defined in terms of the endpoints of each processor's section of the array. Let $a_0, a_1, \ldots, a_P$ be a set of integers such that $a_0 = 0$, $a_P = N$, and $a_i < a_{i+i}$ for all i. Then a generalized block distribution can be defined as

$$local(p) = \{ i \mid a_p \leq i < a_{p+1} \} \tag{3.7}$$

Block distributions choose the a_is to be as evenly distributed as possible. If the computations required for different points vary, it may be advantageous to select partitions of varying sizes.

3.2.3 Multi-Dimensional Distribution Patterns

The simplest distribution patterns for multi-dimensional arrays are obtained by applying one-dimensional distribution patterns to a single dimension and not distributing any other dimension. Kali distributions of multi-dimensional data arrays on one-dimensional processor arrays are examples

of this. In these cases, one dimension is essentially ignored in the definition of *local*. For example, the *local* functions for the D and E arrays declared in Figure 2.2 of Section 2.1.3 are

$$local_D(p) = \{ (i,j) \mid i \equiv p \pmod{P} \} \tag{3.8}$$

$$local_E(p) = \left\{ (i,j) \;\middle|\; \frac{M}{P}(p-1)+1 \leq j < \frac{M}{P}p+1 \right\} \tag{3.9}$$

Adding and subtracting one in Equations 3.9 is required by the 1-based indexing of the processor and data arrays.

A generalization of the above approach is to distribute several dimensions independently on a multi-dimensional processor set. The multi-dimensional Kali distributions in Figure 2.8 in Section 2.1.3 are examples of this. These distributions generally give rise to several independent conditions in definitions of *local* functions. For example, the *local* functions for the arrays of Figure 2.8 are

$$local_F(p_1,p_2) = \left\{ (i,j) \;\middle\|\; \frac{M}{P}\cdot(p_1-1)+1 \leq i < \frac{M}{P}\cdot p_1+1 \text{ and } \frac{M}{P}\cdot(p_2-1)+1 \leq j < \frac{M}{P}\cdot p_2+1 \right\} \tag{3.10}$$

$$local_G(p_1,p_2) = \left\{ (i,j) \;\middle\|\; i \equiv p_1 \pmod{P} \text{ and } \frac{M}{P}\cdot(p_2-1)+1 \leq j < \frac{M}{P}\cdot p_2+1 \right\} \tag{3.11}$$

Again, the dimension-wise formulas must be adjusted from 0-based to 1-based indexing.

The two-dimensional block distribution of Equation 3.10 is often extended with overlaps between adjacent processors to reduce communication. The *local* function for this modification is

$$local(p_1,p_2) = \left\{ (i,j) \;\middle\|\; \frac{N_1}{P_1}\cdot p_1-1 \leq i < \frac{N_1}{P_1}\cdot(p_1+1)+1 \text{ and } \frac{N_2}{P_2}\cdot p_2-1 \leq j < \frac{N_2}{P_2}\cdot(p_2+1)+1 \right\} \tag{3.12}$$

Figure 3.4 illustrates this pattern for $N_1 = N_2 = 8$, $P_1 = P_2 = 2$. Processors $(0,0)$ and $(1,1)$ are shown using dotted outlines there to highlight the overlap. Overlaps larger than 1 can also be used; their *local* functions are similar to the above.

The skewed distribution pattern is often used to pipeline computations on a two-dimensional matrix. The basic idea is to give each processor several "slices" of the array parallel to the main diagonal. Figure 2.12 illustrates this idea for $N_1 = N_2 = 8$ and $P = 4$. This is essentially the same distribution that was used in Figure 1.4; a user-defined Kali distribution for it was shown in Figure 2.11. The relevant *local* function is

$$local(p) = \{ (i,j) \mid i-j \equiv p \pmod{P} \} \tag{3.13}$$

This function is independent of whether the data and processor arrays use 1-based or 0-based indexing.

Many other multi-dimensional distributions are possible. Chueng and Reeves [CR89] suggest dividing two-dimensional domains into irregularly placed rectangular blocks. If the ith block is described by its lower left corner (l_i, b_i) and upper right corner (r_i, t_i), then an appropriate *local* function is

$$local(p) = \{ (i,j) \mid l_i \leq i \leq r_i \text{ and } b_i \leq j \leq t_i \} \tag{3.14}$$

Snyder and Socha [SS90] describe a scheme for generating partitions that are "balanced, near-rectangular, and near-bulky." Each of their partitions can be described as the union of at most

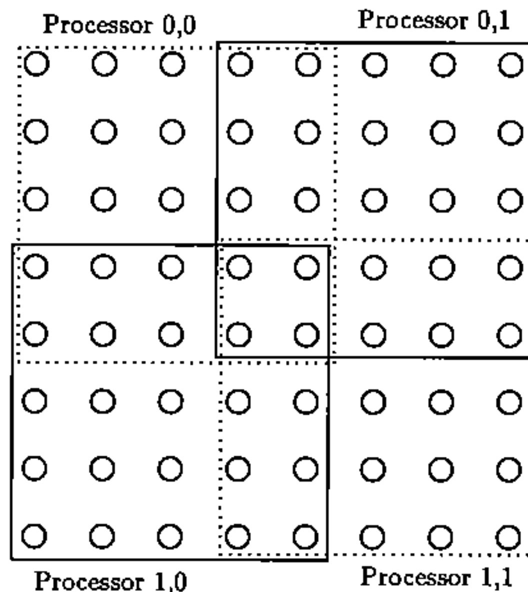


Figure 3.4: Two-dimensional block distribution with overlap

13 canonical rectangles. A *local* function describing such a partition would involve the union of 13 expressions like the right-hand side of Equation 3.14. Other distribution patterns can be described in similar terms. We close this section by noting that, although the examples here have focused on two-dimensional arrays, similar techniques can be applied to arrays with with three or more dimensions.

3.2.4 Other Distributions

The last two subsections have considered data distributions which can be expressed analytically. In this subsection we turn our attention to other distributions.

In some applications it is necessary to distribute an array based on the run-time values stored in that array or in another related array. Such data distributions are known as dynamic distributions, and are often used to balance the computational load or to minimize communications among the processors. Many examples of this technique are given in [FJL+86]. The data distribution can still be modeled by *local* functions in these cases, but more complex definitions of those functions must be used. In particular, the *local* function will depend on program data, and cannot be evaluated before the program is executed. Implementing such distributions efficiently is quite complex because no information is available to the compiler.

Finally, there are large data structures which are not arrays, such as trees and linked lists. These structures must also be distributed across processor memories for the same reasons that arrays are. In these cases, however, defining the distribution may be somewhat harder than for arrays. There are two reasons for this:

1. These data structures are typically built dynamically, and thus require dynamic data distributions such as those described above.

2. Many of these structures do not have a simple identification scheme for their components. This makes it notationally difficult to describe mappings between those components and the processor set.

Because of these complexities, we do not discuss non-array data structures in this thesis. Further work is needed to implement programs using those data structures on nonshared memory parallel machines.

3.3 Parallel Loops

We now turn to modeling the distribution of **forall** iterations on a nonshared memory computer. This is similar to the data distribution described in Section 3.2, except that computation is being distributed rather than data. A mathematical model of this distribution of computation follows the same lines as the data distribution model given above.

Definition 3.2 *Let Procs be the set of processors, and Iter the set of iterations of a* **forall** *statement. Then*

$$exec : Procs \rightarrow 2^{Iter} : exec(p) = \{\, i \in Iter \mid i \text{ is executed by } p \,\} \tag{3.15}$$

Iter is precisely the range given in the **forall** statement, which may depend on bounds computed at run time. We assume that it is represented by the values of the **forall** index variable. For simplicity of presentation, we will only refer to a single **forall** in most of our examples.

As with the *local* functions described earlier, the *exec* formalism allows one iteration to be executed by more than one processor. In practice, this is not common; the only situation where it may be useful is to avoid communication by duplicating a computation on several processors. In what follows, we will assume that the *exec* sets of differing processors are disjoint. In practice, computations are often distributed to the processor where the necessary data is stored. Thus, when a **forall** iterates over the elements of array A we have $exec(p) = local_A(p)$. Often only a section of the array is manipulated, in which case the weaker statement $exec(p) \subseteq local_A(p)$ applies. Because of this close relationship, there is little purpose in showing examples of *exec* functions as we did for *local* functions.

3.4 Communication Sets

We now turn our attention to the communication sets. The purpose of this section is to define and derive expressions for $send_set(p,q)$ and $recv_set(p,q)$ in terms of the *local* and *exec* functions. This differs from the last two sections, in which we only defined the functions. This is because the *local* and *exec* functions were defined by the Kali program text; the communication sets require more work to derive. We will not discuss the practicalities of computing the sets here. Details of implementing this generation will be discussed in Chapters 4 and 5.

The information needed to generate messages in the implementation of a Kali **forall** can be encapsulated in the two new set-valued functions given in Definition 3.3.

Definition 3.3 *Let Procs be the set of processors, and Elem the set of elements of array A. Then*

$$send_set : Procs \times Procs \rightarrow 2^{Elem} :$$
$$send_set(p,q) = \{\, a \in Elem \mid a \text{ must be sent from } p \text{ to } q \,\} \tag{3.16}$$
$$recv_set : Procs \times Procs \rightarrow 2^{Elem} :$$
$$recv_set(p,q) = \{\, a \in Elem \mid a \text{ must be received by } p \text{ from } q \,\} \tag{3.17}$$

Note that $send_set(p,q) = recv_set(q,p)$ for all p and q, reflecting the fact that every message has a sender and a receiver.

We first derive an expression for $recv_set(p,q)$. An array element e must be in $recv_set(p,q)$ if two conditions are met:

1. Processor q must store e. Otherwise, q cannot access e to send it.

2. Processor p must access e. Otherwise, there is no reason to receive e.

The first condition is satisfied by $e \in local(q)$. To represent the second condition, we define a new set function.

$$ref : Procs \rightarrow 2^{Elem} : ref(p) = \{\, e \in Elem \mid p \text{ accesses } e \,\} \tag{3.18}$$

Under the simplifying assumptions given for Figure 3.1, the only way for processor p to access array element e is for $e = A[f(i)]$ for some $i \in exec(p)$. Thus, we can give a simple formula for $ref(p)$:

$$\begin{aligned} ref(p) &= \{\, f(i) \in Elem \mid i \in exec(p) \,\} \\ &= f(exec(p)) \end{aligned} \tag{3.19}$$

Combining the two conditions stated above, we have that $e \in recv_set(p,q)$ if $e \in local(q)$ and $e \in ref(p)$ or, equivalently,

$$\begin{aligned} recv_set(p,q) &= local(q) \cap ref(p) \\ &= local(q) \cap f(exec(p)) \end{aligned} \tag{3.20}$$

We will take this as the general formula for $recv_set(p,q)$.

The above analysis of $recv_set(p,q)$ can be visualized easily for **block** distributions in two dimensions, as shown in Figure 3.5. The diagram represents a portion of the array space used in the Kali program above it, where each circle represents one element of array A. The $local(p)$ sets are squares in the data space. Because of the form of the **on** clause, the $exec(p)$ sets are the same as the $local(p)$ sets. These sets are represented by the large dashed rectangles in the figure. Only four of these sets are shown; the other data elements are contained in the $local(q)$ sets for other processors q. Subscripting functions shift and deform these rectangles to produce the $ref(p)$ sets. The common case of linear subscript functions f deforms the $exec(p)$ sets into parallelograms and shifts them to new positions. Here the subscripting function is $f(i,j) = \langle i-1, j-1 \rangle$, which shifts the squares up and to the left and does not deform them. One such set is shown as the dotted square in the figure. Intersections between the sets are shown as solid rectangles. In this case there are three nonempty $recv_set(p,q)$ sets for each processor p.

The analysis of the $send_set$ function mirrors that of $recv_set$. In this case, the conditions on an element $e \in send_set(p,q)$ are

1. Processor p must store e. Otherwise, p cannot access e to send it.

2. Processor q must access e. Otherwise, there is no reason to receive e.

These conditions lead directly to the expression for $send_set(p,q)$:

$$\begin{aligned} send_set(p,q) &= local(p) \cap ref(q) \\ &= local(p) \cap f(exec(q)) \end{aligned} \tag{3.21}$$

Comparing Equations 3.20 and 3.21 we observe a fundamental relationship between $send_set$ and $recv_set$:

$$send_set(p,q) = recv_set(q,p) \tag{3.22}$$

This is, of course, to be expected since every message received by one processor must be sent by another. The analysis for $send_set(p,q)$ can be visualized in the same way that the analysis for $recv_set(p,q)$ can. This is why Figure 3.5 labels the communication sets as both $send_set$ and $recv_set$.

```
var A : array[ 1..N, 1..N ] of real dist by [ block, block ] on Procs;

forall i in 2..N, j in 2..N on A[i,j].loc do
        ⋮
   ... A[i−1,j−1] ...
        ⋮
end;
```

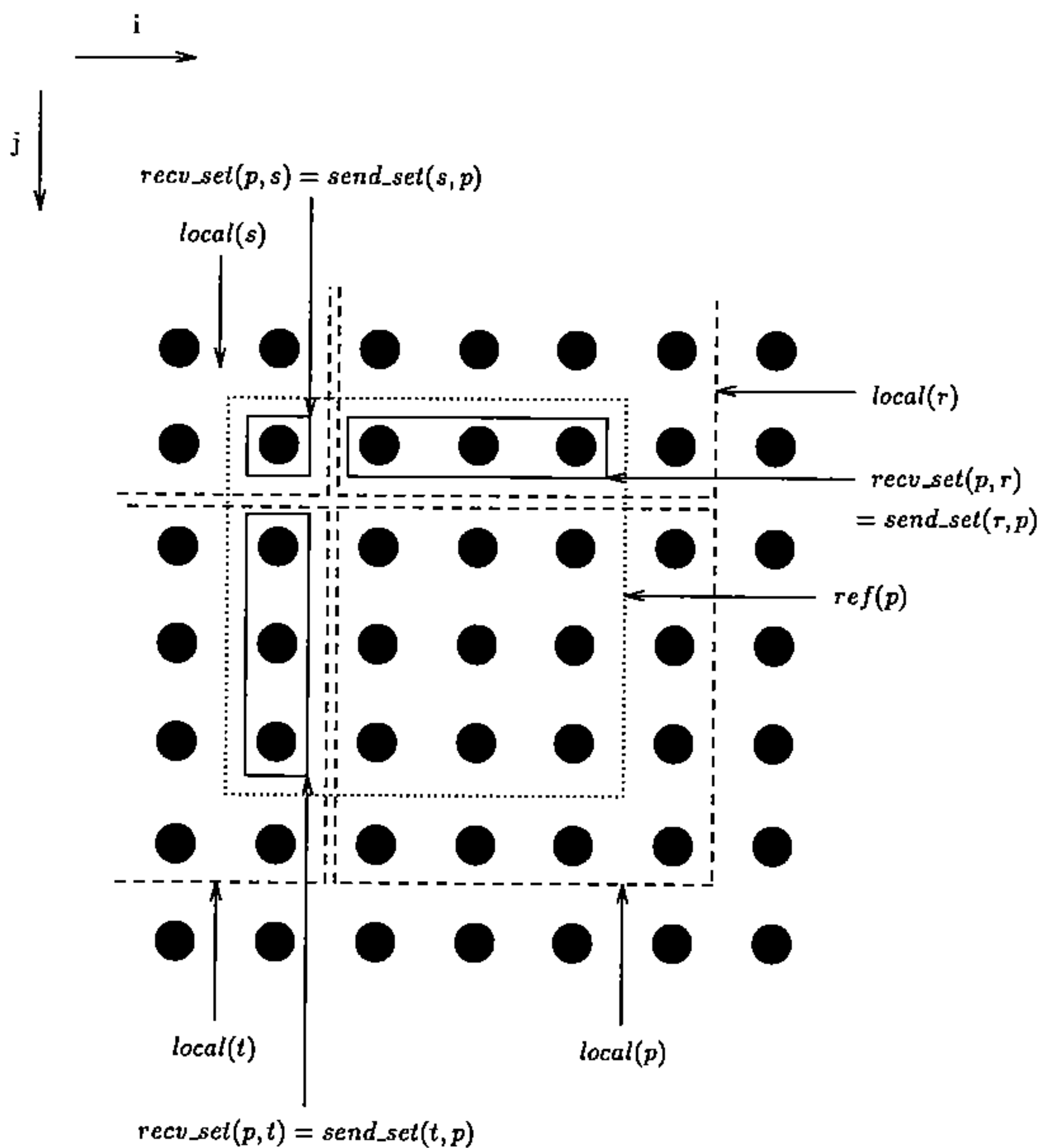


Figure 3.5: Visualizing communication sets

3.5 Iteration Sets

The purpose of this section is to derive expressions for the iteration sets in the same way that the last section derived expressions for the communication sets. We will derive expressions for $local_iter(p)$ and $nonlocal_iter(p)$ without discussing the practicalities of their implementation. Chapters 4 and 5 will use these expressions as the basis of their analysis.

Formally, the iteration sets are defined as two set-valued functions.

Definition 3.4 *Let Procs be the set of processors, and Iter the set of iterations of a* **forall**. *Then*

$$local_iter(p) : Procs \rightarrow 2^{Iter} :$$
$$local_iter(p) = \{ i \in exec(p) \mid i \text{ uses only data on } p \} \quad (3.23)$$
$$nonlocal_iter(p) : Procs \rightarrow 2^{Iter} :$$
$$nonlocal_iter(p) = \{ i \in Iter \mid i \text{ uses some data not on } p \} \quad (3.24)$$

Note that both iteration sets are subsets of $exec(p)$.

As was the case for $recv_set(p, q)$, two conditions must be satisfied in order for an iteration i to be in $local_iter(p)$:

1. Processor p must execute i.

2. Iteration i must access only data stored on p.

The first condition is satisfied when $i \in exec(p)$. To represent the second condition, we define a new set function.

$$deref(p) : Procs \rightarrow 2^{Iter} : deref(p) = \{ i \in Iter \mid i \text{ accesses only data on } p \} \quad (3.25)$$

Although their definitions are very similar, it is not the case that $deref(p)$ is the same as $local_iter(p)$. The difference is that $deref(p)$ may include iterations not executed on processor p. For example, if all iterations of the **forall** accessed the same array element e (and no other elements), then $deref(p)$ would be all of $Iter$ for the processor storing e. For the program of Figure 3.1, there is only one way that any array element e can be accessed: if $e = A[f(i)]$ for some iteration i. Thus, we can derive a simple formula for $deref(p)$:

$$\begin{aligned} deref(p) &= \{ i \in Iter \mid f(i) \in local(p) \} \\ &= f^{-1}(local(p)) \end{aligned} \quad (3.26)$$

Combining the above two conditions, we have that $i \in local_iter(p)$ if $i \in exec(p)$ and $i \in deref(p)$ or, equivalently,

$$\begin{aligned} local_iter(p) &= exec(p) \cap deref(p) \\ &= exec(p) \cap f^{-1}(local(p)) \end{aligned} \quad (3.27)$$

This is the general formula for $local_iter(p)$ that we sought. Since iterations on processor p which do not fall in $local_iter(p)$ must fall into $nonlocal_iter(p)$, we can define $nonlocal_iter(p)$ by set complement:

$$nonlocal_iter(p) = exec(p) - local_iter(p)$$

Rewriting and simplifying, we find

$$\begin{aligned} nonlocal_iter(p) &= exec(p) - local_iter(p) \\ &= exec(p) - (exec(p) \cap deref(p)) \\ &= (exec(p) - exec(p)) \cup (exec(p) - deref(p)) \\ &= \phi \cup (exec(p) - deref(p)) \\ &= exec(p) - deref(p) \end{aligned}$$

We take the last form as our definition of *nonlocal_iter*.

$$\begin{aligned} nonlocal_iter(p) &= exec(p) - deref(p) \\ &= exec(p) - f^{-1}(local(p)) \end{aligned} \quad (3.28)$$

The iteration sets can be visualized for two-dimensional **block** distributions in much the same way as the communication sets, as shown in Figure 3.6. Here, the squares represent **forall** iterations rather than array elements. One $exec(p)$ set is shown as a dashed rectangle; because of the **on** clause in the **forall**, this is the same as the $local(p)$ set. The inverses of subscript functions deform this set in much the same way that subscript functions did in the last section. In this case, $f(i,j) = \langle i-1, j-1\rangle$, so $f^{-1}(i,j) = \langle i+1, j+1\rangle$ and the effect of f^{-1} is to shift the $local(p)$ set down and to the right. This produces $deref(p)$, shown as a dotted square. The $local_iter(p)$ and $nonlocal_iter(p)$ sets are shown as the solid square and solid L-shaped region, respectively.

3.6 Extensions to the Example

The above discussion, particularly the outline in Figure 3.2 and Equations 3.19 through 3.28, makes several assumptions about the program being analyzed (Figure 3.1). This section discusses changes needed to relax the following three assumptions:

1. $A[f(i)]$ is the only array reference in the loop.
2. $A[f(i)]$ is an r-value rather than an l-value, that is, $A[f(i)]$ is not the target of an assignment.
3. $A[f(i)]$ is always accessed in the loop, that is, there are no conditionals to alter control flow around the reference.

We do not consider other possible generalizations of our methods, such as their application to other types of parallel loops. These will be touched on briefly in Chapter 7.

Allowing several array references in a **forall** does not change the basic analysis of either the communication or the iteration sets. For example, an array element must be received by processor p from processor q if it is accessed by p and stored by q, regardless of the number of program expressions causing that access. The expressions for $ref(p)$ and $deref(p)$ do need changing for this case, however. Since $ref(p)$ is the set of all elements accessed by p, it must be the union of the elements accessible by each individual program expression. Similarly, $deref(p)$ is the intersection of the iterations that access only local array elements using each expression. This can be represented as

$$ref(p) = \bigcup_k f_k(exec(p)) \quad (3.29)$$

$$deref(p) = \bigcap_k f_k^{-1}(local(p)) \quad (3.30)$$

where an arbitrary array reference in the program is $A[f_k(i)]$. Similarly, if several different arrays are accessed, the elements of $ref(p)$ must be tagged to differentiate between arrays, and the correct *local* function for each array must be used in the definition of $deref(p)$. Equations 3.20, 3.21, 3.27, and 3.28 remain correct if Equations 3.29 and 3.30 are used to define $ref(p)$ and $deref(p)$.

Assignment to $A[f(i)]$ requires two new communication sets, and two extra steps in the outline of Figure 3.2. Because a reference to $A[f(i)]$ now implies that that array element is written rather than read, the direction of the communications is reversed. If processor p assigns to an array element stored on processor q, then p must send a message and q must receive it. This leads to the definitions of two new set functions:

$$\begin{aligned} & assign_send_set : Procs \rightarrow 2^{Elem} : \\ & \quad assign_send_set(p,q) = \{\, e \in Elem \mid p \text{ assigns } e \text{ and } q \text{ stores } e \,\} \quad (3.31) \\ & assign_recv_set : Procs \rightarrow 2^{Elem} : \\ & \quad assign_recv_set(p,q) = \{\, e \in Elem \mid p \text{ stores } e \text{ and } q \text{ assigns } e \,\} \quad (3.32) \end{aligned}$$

```
var A : array[ 1..N, 1..N ] of real dist by [ block, block ] on Procs;

forall i in 1..N, j in 1..N on A[i,j].loc do
        ⋮
  ... A[i−1,j−1] ...
        ⋮
end;
```

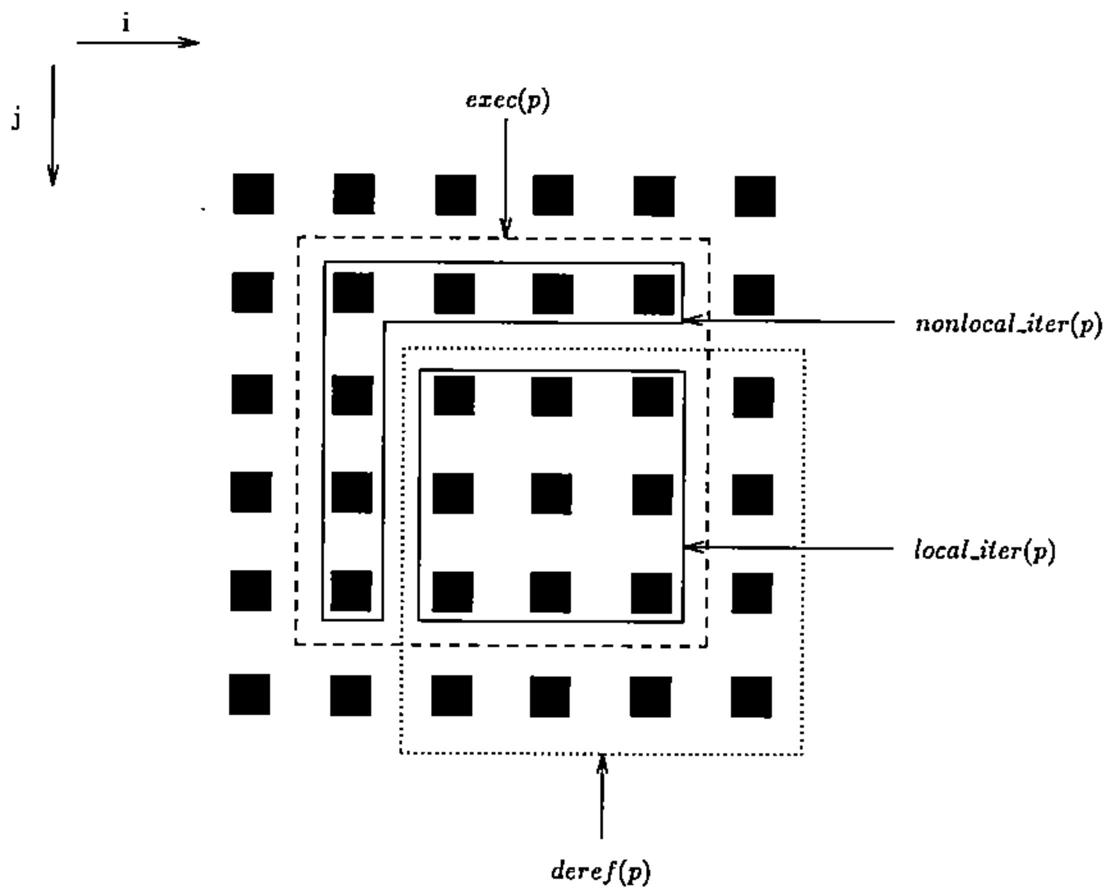


Figure 3.6: Visualizing iteration sets

Code on processor p:

- Generate communication and iteration sets
  - $local(p)$ = Array elements stored on p.
  - $exec(p)$ = Iterations to be performed on p.
  - For all $q \neq p$, $send_set(p,q)$ = Array elements sent from p to q.
  - For all $q \neq p$, $recv_set(p,q)$ = Array elements received by p from q.
  - For all $q \neq p$, $assign_send_set(p,q)$ = Elements on q that p assigns.
  - For all $q \neq p$, $assign_recv_set(p,q)$ = Elements on p that q assigns.
  - $local_iter(p)$ = Iterations on p that access only local data.
  - $nonlocal_iter(p)$ = Iterations on p that access some nonlocal data.
- For all q with $send_set(p,q) \neq \phi$, send message containing $send_set(p,q)$ to q.
- Execute computations for iterations in $local_iter(p)$, accessing only local arrays.
- For all q with $recv_set(p,q) \neq \phi$, receive message with $recv_set(p,q)$ from q.
- Execute computations for iterations in $nonlocal_iter(p)$, accessing local arrays and message buffers.
- For all q with $assign_send_set(p,q) \neq \phi$, send message containing $assign_send_set(p,q)$ to q.
- For all q with $assign_recv_set(p,q) \neq \phi$, receive message with $assign_recv_set(p,q)$ from q and perform assignments to local section of array.

Figure 3.7: Adding nonlocal assignments to Figure 3.2

An analysis parallel to that for *send_set* and *recv_set* yields the following expressions:

$$assign_send_set(p,q) = local(q) \cap f(exec(p)) \quad (3.33)$$

$$assign_recv_set(p,q) = local(p) \cap f(exec(q)) \quad (3.34)$$

(Note the similarity of Equations 3.33 and 3.34 to Equations 3.21 and 3.20.) Figure 3.7 shows how these sets are used in the implementation. The initial part of the computation follows Figure 3.2. After that computation finishes, $assign_send_set(p,q)$ is used to send nonlocal array elements that have been assigned to their home processors, and $assign_recv_set(p,q)$ controls receiving elements from other processors. Parallel accumulations onto nonlocal variables can be handled in a similar way; the essential change necessary is to send the values to be accumulated rather than the full accumulated value. If different types of accumulations are done, either new analogues of the communication sets are needed for each accumulation type or flags must be maintained describing the operation to be performed on each communicated datum. A similar statement applies to mixing pure assignments and accumulations.

In theory, conditionals and other control-flow constructs in a **forall** can be handled by applying operators to the $exec(p)$ sets to exclude iterations which do not perform a given array reference. This is seldom a practical approach, however. A more reasonable alternative is to compute *conservative approximations* of the communication and iteration sets. In compiler terminology, a conservative approximation to a set X is another set Y which produces correct results when Y is substituted for X. For example, a vectorizing compiler can transform a loop into a vector instruction only if there are no data dependences in the loop. If the compiler's dataflow analysis cannot prove the lack of a dependence, it must assume the possible dependence exists and not vectorize the code. In other

words, the compiler must conservatively approximate the set of data dependences by assuming it is as large as possible.

In the context of foralls with nested control flow, conservative approximations are needed for the communication and iteration sets. Conservative approximations $send_set'(p,q)$ and $recv_set'(p,q)$ to the communication sets must satisfy three conditions.

$$send_set(p,q) \subseteq send_set'(p,q) \tag{3.35}$$
$$recv_set(p,q) \subseteq recv_set'(p,q) \tag{3.36}$$
$$send_set'(p,q) = recv_set'(q,p) \tag{3.37}$$

Relations 3.35 and 3.36 follow from the observation that a program will still be correct if it performs more communication than absolutely necessary. The last condition is necessary because the program will deadlock if senders and receivers do not agree on the contents of a message (or whether a message should be sent at all). For the iteration sets, the conditions on conservative approximations $local_iter(p)$ and $nonlocal_iter(p)$ are slightly different.

$$local_iter'(p) \subseteq local_iter(p) \tag{3.38}$$
$$nonlocal_iter(p) \subseteq nonlocal_iter'(p) \tag{3.39}$$
$$local_iter'(p) \cap nonlocal_iter'(p) = \phi \tag{3.40}$$
$$local_iter'(p) \cup nonlocal_iter'(p) = exec(p) \tag{3.41}$$

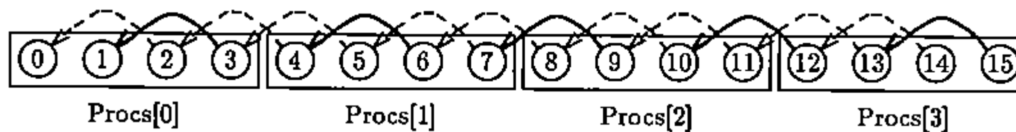
Relations 3.38 and 3.39 hold because a local iteration can execute either before or after messages are received, but a nonlocal iteration must wait for data to arrive. Thus, it is correct to execute a local iteration with the nonlocal iterations but not vice versa; equivalently, it is correct to place a local iteration in $nonlocal_iter'(p)$ but not to place a nonlocal iteration in $local_iter'(p)$. Equations 3.40 and 3.41 guarantee that each iteration is executed exactly once.

When a **forall** contains a conditional statement, the above conservation properties can be maintained by computing the sets assuming that all array references in the conditional are actually executed. This means that Equations 3.20, 3.21, 3.27, and 3.28 are applied without considering whether control flow on a given iteration will reach the array reference. Note that this differs from the behavior defined by Definitions 3.3 through 3.4, which implicitly consider only references which are actually used. Figure 3.8 illustrates this by showing both the actual communication sets and conservative approximations to them in the presence of a conditional. Nested **for** and **while** statements are handled similarly, by assuming that the loop bodies are always executed. In the case of a nested **for** statement, array references often depend on the loop index; this must also be incorporated in computing the communication sets. Essentially, this is done by considering a single reference in a nested for loop as many separate references parameterized by the index. Figure 3.9 gives an example of this.

The three generalizations given in this section are almost independent, in that they may be combined without adverse effects. The only exception to this is combining nonlocal assignments with control-flow constructs. In this case, flags must be used to determine which array elements are actually assigned in the loop. Otherwise, good data values may be overwritten. Note that these flags must be calculated on the assigning processor and used on the storing processor, so they must be communicated along with the array elements.

```
-- assume N=16 and P=4
processors Procs : array[ 0..P-1 ] with P in 1..max_procs;
var A, B : array[ 0..N-1 ] of real dist by [ block ] on Procs;
forall i in 0..N-3 on A[i].loc do
   if ( i%3 = 1 ) then
         A[ i ] := B[ i+2 ];
   end;
end;
```

(a) Kali forall statement with nested conditional



(b) Actual (solid arrows) and assumed (dashed arrows) flow of data

$$recv_set(1,2) = send_set(2,1) = \{9\}$$
$$recv_set(2,3) = send_set(3,2) = \{12\}$$

(c) Exact sets (all other communication sets are empty)

$$recv_set'(0,1) = send_set'(1,0) = \{4,5\}$$
$$recv_set'(1,2) = send_set'(2,1) = \{9,10\}$$
$$recv_set'(2,3) = send_set'(3,2) = \{12,13\}$$

(d) Conservative approximations (all other approximating sets are empty)

Figure 3.8: Conservative approximations of communication sets

```
-- assume N=16 and P=4

processors Procs : array[ 0..P-1 ] with P in 1..max_procs;

var A, B : array[ 0..N-1, 0..M ] of real dist by [ block, * ] on Procs;

forall i in 0..N-3 on A[i].loc do
   for j in 0..M do
        A[i,j] := B[i+2,j];
   end;
end;
```

(a) Kali **forall** statement with nested **for**

$$\begin{aligned} recv_set(0,1) = \quad send_set(1,0) = \quad & \{4,5\} \times \{0,1,\ldots,M\} \\ recv_set(1,2) = \quad send_set(2,1) = \quad & \{8,9\} \times \{0,1,\ldots,M\} \\ recv_set(2,3) = \quad send_set(3,2) = \quad & \{12,13\} \times \{0,1,\ldots,M\} \end{aligned}$$

(b) Communication sets for (a)

Figure 3.9: Communication sets for a **forall** with a nested **for**

3.7 Generating the Sets: Compile-time versus Run-time

The preceding discussion has been necessarily vague, since the concepts here can be applied to any **forall** statement. Performing these transformations on a particular **forall** raises certain new issues:

1. *When* can the required sets be generated?

2. *How* can the required sets be generated?

3. *What representation* should be used for the sets?

Different loops may require different answers to these questions. We have developed two styles of analysis which are applicable to different types of loops: a compile-time analysis useful for regular problems, and a run-time analysis useful for irregular problems. In this section, we discuss general differences between the two methods; the next two chapters will give specifics on their implementations.

Many scientific applications have very regular array access patterns. These access patterns may arise from either the underlying physical domain being studied or the algorithm being used. Examples of such applications include

1. Relaxation algorithms on regular meshes

2. Alternating Direction Implicit (ADI) algorithms

3. Traditional linear algebra operators, such as matrix multiplication

4. Dense matrix factorizations, such as Gaussian elimination

The distribution and subscripting functions used in such applications tend to be simple: some type of block or cyclic distribution, and linear subscript functions. With such functions, the communication and iteration sets can often be described by a few scalar parameters (such as low and high bounds

on a range). Such a representation is very space-efficient and can be calculated quickly if analytic expressions are available for the parameters. Often the compiler can perform these calculations completely or partially; in those cases, the first phase of the code in Figure 3.2 becomes either a set of constant declarations or a few integer operations. We refer to the analysis for these cases as compile-time analysis, discussed in depth in Chapter 4. Compile-time analysis leads to extremely efficient programs, but the price paid is some loss of generality. If no simple descriptions of the sets are obtainable, then this style of analysis does not apply.

In applications using complex distributions or subscripts, it may be impossible to give any analytic expressions describing the communication and iteration sets. This is a particularly serious problem if the subscript functions depend on data only available at run-time, such as index arrays. Examples of applications in this class include

1. Relaxation algorithms on unstructured meshes

2. Sparse linear algebra methods, which use specialized data structures

3. Sparse matrix factorizations, including incomplete factorization algorithms

Because of the dependence on run-time data to describe the communication and iteration sets for these applications, little can be done at compile-time. In such cases, the compiler must generate code which calculates the sets at run-time. Generally, no succinct description of the sets is available, so some general mechanism for representing sets must be used. We refer to the analysis necessary in this case as run-time analysis and discuss it in depth in Chapter 5. It has the advantage of being quite general, although the code necessary to generate the sets is much more involved than that for compile-time analysis. Chapter 6 shows that this added overhead can be made acceptably small.

Chapter 4

Compile-time Analysis

In this chapter we develop formulas which can be used in the compiler to represent the communication and iteration sets defined in Chapter 3. Using these formulas, the compiler can emit code to generate the program described in Section 3.1. In order for the compiler to do this, several things must be true:

1. The *local* functions for the arrays must be easily represented in the compiler. In this chapter, we will only consider the Kali **block** and **cyclic** distributions.

2. The *exec* functions must have a similarly simple form. In this chapter, we will assume that the *exec* function is equivalent to the *local* function for some array accessed by the **forall**.

3. The subscripts must be relatively simple. If a subscript depends on an array known only at run time, for example, the compiler will not be able to analyze the reference. In this chapter, we will consider subscripts which represent linear functions of the **forall** loop index.

The above conditions may appear very restrictive, but in practice they cover many of the most common cases. Nearly all of the applications cited in [FJL+86] use **block** or **cyclic** distributions of the data. In the same reference, the assumption is usually made that computations are done on the processor where the result is to be stored, which generally satisfies our second criteria. Finally, a recent study [SLY88] indicates that as many as 78% of all subscripts in the scientific codes studied were linear functions of loop indices or could be converted to linear subscripts using user assertions. Thus, even a restricted compile analysis in the compiler will be widely applicable.

The precise conditions that we impose on subscripts are worth noting. In Sections 4.2 through 4.4, we will declare several variables used in subscript expressions as constants. We assume these constants are integers. This is natural, since the subscript itself must be an integer. The Kali compiler does not require constants in actual programs, however. Instead, it allows any expression which is invariant during the **forall** statement to be used. Such invariant expressions are detected by standard compiler techniques. Figure 4.1 shows several examples of these. The references $B[c]$, $B[j]$, $B[k]$ and $B[j+k]$ are **forall**-invariant and can all be handled by the analysis in Section 4.2. Reference $B[bad]$, however, is outside the scope of our analysis because the value of *bad* changes within the same iteration of the **forall**. This is more general than the definition of linear subscript used in [SLY88], which requires coefficients to be known at compile time. Our results may therefore be even more general than the comments above indicate.

The remainder of this chapter is organized as follows. Section 4.1 introduces some notation that will be used in the other sections. Section 4.2 describes the compiler analysis possible when the subscript is a constant; this analysis applies to Kali's **block**, **cyclic**, and even user-defined distributions. Section 4.3 describes the compiler analysis possible for one-dimensional arrays distributed by **block**. Section 4.4 describes the compiler analysis for one-dimensional **cyclic** arrays. Finally, Section 4.5 describes some extensions to the results of the other sections.

```
processors procs : array[ 1..NP ] with NP in 1..max_procs;

const
   c = 10;

var
   A, B : array[ 1..N ] of real dist by [ block ] on procs;


for j in 1..10
   var k : integer;
do
   k := (j*j + j) / 2;
   forall i in low..high on A[i].loc
         var bad : integer;
   do
         bad := round( sqrt( i ) );
         A[i] := B[ c ];      -- OK; compile-time constant
         A[i] := B[ j ];      -- OK; forall loop invariant treated as constant
         A[i] := B[ k ];      -- OK; forall loop invariant treated as constant
         A[i] := B[ j+k ];           -- OK; value of j+k is loop invariant
         A[i] := B[ bad ];           -- Trouble; not forall loop invariant
   end;
end;
```

Figure 4.1: Forall-invariant subscripts

4.1 Notation

This section introduces a notation for describing ranges of integers, which will appear in the analysis later in this chapter. Proofs of the lemmas presented here are omitted, as they are tangential to the main thrust of this chapter. They may be found in [Koe90].

Definition 4.1 *A contiguous range of integers is denoted by*

$$[A, B] = \{ i \mid A \leq i \leq B \} \tag{4.1}$$

Non-contiguous ranges with a constant (integer) step size are denoted by

$$[A,\ B;\ C] = \{ i \mid A \leq i \leq B \wedge i \equiv A \pmod{C} \} \tag{4.2}$$

with C restricted to be positive.

In order to describe certain properties of ranges, it is convenient to define the following:

Definition 4.2 *For integer a and b and positive integer c, let $nxt(a, b, c)$ be the smallest integer such that $nxt(a, b, c) \geq a$ and $nxt(a, b, c) \equiv b \pmod{c}$.*

Lemma 4.1 tells how this function can be computed.

Lemma 4.1 *If the modulo operator % is defined so that $a \,\%\, b \geq 0$ for all a and b, then*

$$nxt(a, b, c) = a + (b - a) \,\%\, c \tag{4.3}$$

Using $nxt(a, b, c)$ we can derive several properties of ranges

```
processors procs : array[ 1..NP ] with NP in 1..max_procs;

const
   low = ...;
   high = ...;
   c = ...;
   N = ...;

var
   A, B : array[ 1..N ] of real dist by [ block ] on procs;

forall i in low..high on A[i].loc do
   A[i] := B[ c ];
end;
```

Figure 4.2: Program using constant subscripts

Lemma 4.2 *For any $c_1 > 0$, $c_2 > 0$, let n_1, n_2 be integers such that $c_1 n_1 + c_2 n_2 = \gcd(c_1, c_2)$ and let*

$$m = nxt\left(\max(a_1, a_2),\ \frac{c_1 n_1 (a_2 - a_1)}{\gcd(c_1, c_2)} + a_1,\ lcm(c_1, c_2)\right)$$

Then

$$\begin{aligned} &[a_1,\ b_1;\ c_1] \cap [a_2,\ b_2;\ c_2] = \\ &\qquad \begin{cases} [m,\ \min(b_1, b_2);\ lcm(c_1, c_2)] & \text{if } a_2 \equiv a_1 \pmod{\gcd(c_1, c_2)} \\ \phi & \text{otherwise} \end{cases} \end{aligned} \tag{4.4}$$

$$\begin{aligned} &[a_1,\ b_1;\ c_1] - [a_2,\ b_2;\ c_2] = \\ &\qquad [a_1,\ \min(b_1, a_2 - 1);\ c_1] \cup [\max(a_1, nxt(b_2 + 1, a_1, c_1)),\ b_1;\ c_1] \cup \\ &\qquad \bigcup_{\substack{0 \le k < c_2/\gcd(c_1, c_2) \\ a_1 + kc_1 \not\equiv a_2 \pmod{c_2}}} [a_1 + kc_1,\ b_1;\ lcm(c_1, c_2)] \end{aligned} \tag{4.5}$$

$$[a_1, b_1] \cap [a_2, b_2] = [\max(a_1, a_2), \min(b_1, b_2)] \tag{4.6}$$

$$[a_1, b_1] - [a_2, b_2] = [a_1, \min(b_1, a_2 - 1)] \cup [\max(a_1, b_2 + 1), b_2] \tag{4.7}$$

$$[a_1, b_1] \cap [a_2,\ b_2;\ c] = [\max(nxt(a_1, a_2, c), a_2),\ \min(b_1, b_2);\ c] \tag{4.8}$$

For convenience we also define one other constant.

Definition 4.3 *If the array being referenced by a* forall *has N elements distributed over P processors, then define*

$$M = \left\lceil \frac{N}{P} \right\rceil \tag{4.9}$$

to be the number of elements stored on each processor.

4.2 Constant Subscripts

We consider the simplest possible case first: arrays indexed by a constant subscript. An example program which falls into this class is shown in Figure 4.2; we will refer to this program throughout

this section. The defining characteristic of this class is that the subscripting function is $f(i) = c$. As discussed earlier, c can be any **forall**-invariant expression. An important example of this class of programs is Gaussian elimination, which we will consider in Chapter 6.

The basic results for this class of subscripts are given in Theorem 4.1.

Theorem 4.1 *If the subscripting function in a* **forall** *is $f(i) = c$ for some invariant c, then*

$$recv_set(p,q) = \begin{cases} \{c\} & \text{if } c \in local(q) \text{ and } exec(p) \neq \phi \\ \phi & \text{otherwise} \end{cases} \quad (4.10)$$

$$send_set(p,q) = \begin{cases} \{c\} & \text{if } c \in local(p) \text{ and } exec(q) \neq \phi \\ \phi & \text{otherwise} \end{cases} \quad (4.11)$$

$$local_iter(p) = \begin{cases} exec(p) & \text{if } c \in local(p) \\ \phi & \text{otherwise} \end{cases} \quad (4.12)$$

$$nonlocal_iter(p) = \begin{cases} \phi & \text{if } c \in local(p) \\ exec(p) & \text{otherwise} \end{cases} \quad (4.13)$$

To prove the theorem, we need the following lemma.

Lemma 4.3 *If the subscripting function in a* **forall** *is $f(i) = c$ for some invariant c, then*

$$ref(p) = \begin{cases} \{c\} & \text{if } exec(p) \neq \phi \\ \phi & \text{if } exec(p) = \phi \end{cases} \quad (4.14)$$

$$deref(p) = \begin{cases} Iter & \text{if } c \in local(p) \\ \phi & \text{if } c \notin local(p) \end{cases} \quad (4.15)$$

As before, Iter is the entire range of the **forall**.

Proof.

$$\begin{aligned} ref(p) &= f(exec(p)) \\ &= \{ f(i) \mid i \in exec(p) \} \\ &= \{ c \mid i \in exec(p) \} \\ &= \begin{cases} \{c\} & \text{if } exec(p) \neq \phi \\ \phi & \text{if } exec(p) = \phi \end{cases} \end{aligned}$$

$$\begin{aligned} deref(p) &= f^{-1}(local(p)) \\ &= \begin{cases} Iter & \text{if } c \in local(p) \\ \phi & \text{if } c \notin local(p) \end{cases} \end{aligned}$$

□

We now return to the proof of Theorem 4.1.

Proof. (Of Theorem 4.1)

All of the formulas are direct applications of Lemma 4.3 to the formulas of Chapter 3.

Equation 4.10:

$$\begin{aligned} recv_set(p,q) &= local(q) \cap ref(p) \\ &= \begin{cases} \{c\} & \text{if } exec(p) \neq \phi \text{ and } c \in local(q) \\ \phi & \text{otherwise} \end{cases} \end{aligned}$$

Equation 4.11:

$$\begin{aligned} send_set(p,q) &= local(p) \cap ref(q) \\ &= \begin{cases} \{c\} & \text{if } exec(q) \neq \phi \text{ and } c \in local(p) \\ \phi & \text{otherwise} \end{cases} \end{aligned}$$

```
-- Code on processor p

const
   low = ...;
   high = ...;
   c = ...;
   low_A = (p-1) * N / P + 1;      -- bounds on local section of A
   high_A = p * N / P;
   low_B = (p-1) * N / P + 1;      -- bounds on local section of B
   high_B = p * N / P;

var A, B : array[ 1..N ] of real dist by [ block ] on procs;
   c : integer;
   temp : real;              -- compiler-generated temporary

-- communication statements
if ( low_B <= c and c <= high_B ) then
   -- broadcast to others
   temp := B[ c ];
   send( temp, procs[*] );
else
   -- receive broadcast
   temp := recv( procs[*] );
end;
-- computation statements
for i in max(low_A,low) .. min(high_A,high) do
   A[i] := temp;
end;
```

Figure 4.3: Compiled form of Figure 4.2

Equation 4.12:

$$\begin{aligned} local_iter(p) &= exec(p) \cap deref(p) \\ &= \begin{cases} exec(p) & \text{if } c \in local(p) \\ \phi & \text{otherwise} \end{cases} \end{aligned}$$

Equation 4.13:

$$\begin{aligned} nonlocal_iter(p) &= exec(p) - deref(p) \\ &= \begin{cases} \phi & \text{if } c \in local(p) \\ exec(p) & \text{otherwise} \end{cases} \end{aligned}$$

□

Notice that the proofs of Lemma 4.3 and Theorem 4.1 use no information concerning the form of either the *local* or the *exec* functions. This indicates that the formulas derived are applicable to any array distributions and any on clauses used. It should also be noted that the expressions for $recv_set(p,q)$ and $send_set(p,q)$ indicate that one processor is sending a value to all other processors. This type of broadcast is precisely the behavior that we expect from a program repeatedly accessing a fixed array element.

Figure 4.3 shows how Theorem 4.1 can be used to implement the program of Figure 4.2. The amount of memory needed to store the received values can be found directly from the maximum size of $recv_set(p,q)$; a scalar variable is sufficient. Because one processor is sending to all others, an efficient broadcast mechanism (either hardware broadcast or a fan-out tree) can be used instead

```
processors procs : array[ 0..P-1 ] with P in 1..max_procs;

const
   N = ...;
   low = ...;
   high = ...;
   c0 = ...;
   c1 = ...;

var
   A, B : array[ 0..N-1 ] of real dist by [ block ] on procs;

forall i in low..high on A[i].loc do
   A[ i ] := B[ c0*i + c1 ];
end;
```

Figure 4.4: Program using linear subscript functions and block array distributions

of sending individual messages. Separate loops for local and nonlocal iterations can be avoided by copying $B[c]$ to the temporary location on the sending processor and considering all iterations on every processor to be nonlocal.

4.3 Block Distributions with Linear Subscripts

We next consider programs such as those shown in Figure 4.4. The defining features of this program are

1. All arrays in the program have a **block** distribution and the same size.
2. The computation is performed on the processor storing element i of one of the arrays.
3. The subscripting function is $f(i) = c_0 i + c_1$, that is, a linear function of the **forall** index. The discussion on page 44 described our assumptions about c_0 and c_1.

Programs with these features include relaxation and ADI algorithms for solving partial differential equations on regular grids. In order to generate useful expressions for the communication and iteration sets, it is necessary to consider two general cases for the value of c_0: $c_0 > 0$ and $c_0 < 0$. (The case of $c_0 = 0$ was handled in Section 4.2.) Theorem 4.2 gives the expressions needed for the $c_0 > 0$ case, while Theorem 4.3 handles the $c_0 < 0$ case. An important subclass of these programs have the additional property that $|c_0| = 1$; therefore, we will derive special forms of all our equations for these cases in Theorems 4.4 and 4.5.

The analysis for all cases requires expressions for $local(p)$ and $exec(p)$. These may be found from the material in Sections 3.2 and 3.3; we repeat them here for ease of reference.

$$\begin{aligned} local(p) &= \{ i \mid Mp \leq i \leq Mp + M - 1 \} \\ &= [Mp, Mp + M - 1] \quad (4.16) \\ exec(p) &= [low, high] \cap local(p) \\ &= [\max(low, Mp), \min(high, Mp + M - 1)] \quad (4.17) \end{aligned}$$

Because the bounds of $exec(p)$ will be used so frequently, we designate names for them.

Definition 4.4 *Define $bot(p)$ and $top(p)$ as*

$$bot(p) = \max(low, Mp) \quad (4.18)$$

$$top(p) = \min(high, Mp + M - 1) \quad (4.19)$$

We begin by deriving expressions for the communication and iteration sets when $c_0 > 0$.

Theorem 4.2 *If all arrays in a* **forall** *are distributed by* **block**, *the subscripting function is* $f(i) = c_0 i + c_1$ *and* c_0 *is a positive integer, then let*

$$\begin{aligned} lb(p,q) &= \max(c_0 bot(p) + c_1, nxt(Mq, c_1, c_0)) \\ ub(p,q) &= \min(c_0 top(p) + c_1, Mq + M - 1) \end{aligned}$$

Then:

If $\lceil \frac{c_1+1}{M} \rceil + c_0 p - 1 \le q \le \lfloor \frac{c_1 - c_0}{M} \rfloor + c_0(p+1)$ *then*

$$recv_set(p,q) = [\, lb(p,q),\ ub(p,q);\ c_0\,] \tag{4.20a}$$

Otherwise,

$$recv_set(p,q) = \phi \tag{4.20b}$$

If $\left\lceil \frac{Mp + c_0 - c_1}{M c_0} \right\rceil - 1 \le q \le \left\lfloor \frac{Mp + M - c_1 - 1}{M c_0} \right\rfloor$ *then*

$$send_set(p,q) = [\, lb(q,p),\ ub(q,p);\ c_0\,] \tag{4.21a}$$

Otherwise,

$$send_set(p,q) = \phi \tag{4.21b}$$

$$local_iter(p) = \left[\max\left(bot(p), \left\lceil \frac{Mp - c_1}{c_0} \right\rceil\right), \min\left(top(p), \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor\right)\right] \tag{4.22}$$

If $c_1 \ge 0$ *then*

$$nonlocal_iter(p) = \left[\max\left(bot(p), \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor + 1\right), top(p)\right] \tag{4.23a}$$

Otherwise, if $c_1 \le (MP - 1)(1 - c_0)$ *then*

$$nonlocal_iter(p) = \left[bot(p), \min\left(top(p), \left\lceil \frac{Mp - c_1}{c_0} \right\rceil - 1\right)\right] \tag{4.23b}$$

Otherwise,

$$nonlocal_iter(p) = \left[\max\left(bot(p), \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor + 1\right), top(p)\right] \cup \left[bot(p), \min\left(top(p), \left\lceil \frac{Mp - c_1}{c_0} \right\rceil - 1\right)\right] \tag{4.23c}$$

Note that the conditions on Equations 4.23a and 4.23b are both true when $c_0 = 1$ *and* $c_1 = 0$. *In this case,*

$$nonlocal_iter(p) = \phi \tag{4.23d}$$

To prove Theorem 4.2 we use the following lemma.

Lemma 4.4 *If all arrays in a* **forall** *are distributed by* **block**, *the subscripting function is* $f(i) = c_0 i + c_1$, *and* $c_0 > 0$, *then*

$$\begin{aligned} ref(p) &= [c_0 bot(p) + c_1,\ c_0 top(p) + c_1;\ c_0] & (4.24) \\ deref(p) &= \left[\left\lceil \frac{Mp - c_1}{c_0} \right\rceil, \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor\right] & (4.25) \end{aligned}$$

Proof.
Equation 4.24:

$$\begin{aligned} ref(p) &= f(exec(p)) \\ &= \{ f(i) \mid i \in exec(p) \} \\ &= \{ c_0 i + c_1 \mid bot(p) \le i \le top(p) \} \\ &= \{ k \mid k = c_0 i + c_1 \wedge bot(p) \le i \le top(p) \} \\ &= \{ k \mid k \equiv c_1 \pmod{c_0} \wedge c_0 bot(p) + c_1 \le c_0 i + c_1 \le c_0 top(p) + c_1 \} \\ &= [c_0 bot(p) + c_1,\ c_0 top(p) + c_1;\ c_0] \end{aligned}$$

Equation 4.25

$$\begin{aligned} deref(p) &= f^{-1}(local(p)) \\ &= \{ i \mid f(i) \in local(p) \} \\ &= \{ i \mid Mp \le c_0 i + c_1 \le Mp + M - 1 \} \\ &= \left\{ i \mid \frac{Mp - c_1}{c_0} \le i \le \frac{Mp + M - 1 - c_1}{c_0} \right\} \\ &= \left\{ i \mid \left\lceil \frac{Mp - c_1}{c_0} \right\rceil \le i \le \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor \right\} \\ &= \left[\left\lceil \frac{Mp - c_1}{c_0} \right\rceil, \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor\right] \end{aligned}$$

The insertion of the floor and ceiling functions is legal because the elements of $deref(p)$ are integers.
□

We now apply Lemma 4.4 to obtain Theorem 4.2.

Proof. (Of Theorem 4.2)

Applying Lemma 4.4 to Equations 3.20 and 3.21 produces

$$\begin{aligned} recv_set(p,q) &= local(q) \cap ref(p) \\ &= [Mq, Mq + M - 1] \cap [c_0 bot(p) + c_1,\ c_0 top(p) + c_1;\ c_0] \\ &= [\max(c_0 bot(p) + c_1, nxt(Mq, c_1, c_0)), \\ &\qquad \min(c_0 top(p) + c_1, Mq + M - 1);\ c_0] \\ send_set(p,q) &= local(p) \cap ref(q) \\ &= [Mp, Mp + M - 1] \cap [c_0 bot(q) + c_1,\ c_0 top(q) + c_1;\ c_0] \\ &= [\max(c_0 bot(q) + c_1, nxt(Mp, c_1, c_0)), \\ &\qquad \min(c_0 top(q) + c_1, Mp + M - 1);\ c_0] \end{aligned}$$

These expressions apply to all values of p and q, but are not in the most efficient form for computation. Many processor pairs will not need to communicate during the computation, and therefore many of the sets will be empty. To avoid explicitly computing these sets, we will now find necessary conditions for $recv_set(p,q) \neq \phi$ and $send_set(p,q) \neq \phi$. These conditions will take the form of a range of values for q for any fixed value of p.

We begin by considering *recv_set*(p,q). A necessary condition for that set to be nonempty is for the range

$$R = [\max(c_0 bot(p) + c_1, Mq), \min(c_0 top(p) + c_1, Mq + M - 1)]$$

to also be nonempty, since it is a superset of *recv_set*(p,q). Since $bot(p) \geq Mp$ and $top(p) \leq Mp + M - 1$, R nonempty implies

$$\begin{aligned} c_0 Mp + c_1 &\leq Mq + M - 1 \\ Mq &\leq c_0(Mp + M - 1) + c_1 \end{aligned}$$

(Other conditions also apply, but these are the only inequalities involving both p and q.) By simplifying these constraints, we can develop bounds on q for a particular processor p. The first constraint produces a lower bound

$$\begin{aligned} c_0 Mp + c_1 &\leq Mq + M - 1 \\ c_0 Mp + c_1 - M + 1 &\leq Mq \\ \frac{c_0 Mp + c_1 - M + 1}{M} &\leq q \\ c_0 p - 1 + \frac{c_1 + 1}{M} &\leq q \end{aligned}$$

while the second produces an upper bound

$$\begin{aligned} Mq &\leq c_0(Mp + M - 1) + c_1 \\ Mq &\leq c_0 Mp + c_0 M - c_0 + c_1 \\ q &\leq \frac{c_0 Mp + c_0 M - c_0 + c_1}{M} \\ q &\leq c_0(p + 1) + \frac{c_1 - c_0}{M} \end{aligned}$$

Since p and q must both be integers, we can further limit the possible range of q using ceiling and floor operators:

$$c_0 p - 1 + \left\lceil \frac{c_1 + 1}{M} \right\rceil \leq q \leq c_0(p + 1) + \left\lfloor \frac{c_1 - c_0}{M} \right\rfloor$$

Only the *recv_set*(p,q) values in this range must be generated; all others are empty sets. A similar analysis identifies nonempty values of *send_set*(p,q). The constraints in this case are

$$\begin{aligned} c_0 Mq + c_1 &\leq Mp + M - 1 \\ Mp &\leq c_0(Mq + M - 1) + c_1 \end{aligned}$$

(Note that these are simply the constraints for *recv_set*(p,q) with the roles of p and q reversed.) The first constraint now gives an upper bound on q

$$\begin{aligned} c_0 Mq + c_1 &\leq Mp + M - 1 \\ c_0 Mq &\leq Mp + M - 1 - c_1 \\ q &\leq \frac{Mp + M - 1 - c_1}{Mc_0} \end{aligned}$$

while the second constraint gives the lower bound

$$\begin{aligned} Mp &\leq c_0(Mq + M - 1) + c_1 \\ Mp &\leq c_0 Mq + c_0 M - c_0 + c_1 \\ Mp - Mc_0 + c_0 - c_1 &\leq c_0 Mq \\ \frac{Mp - Mc_0 + c_0 - c_1}{Mc_0} &\leq q \\ \frac{Mp + c_0 - c_1}{Mc_0} - 1 &\leq q \end{aligned}$$

As before, we can apply floor and ceiling operators to obtain integer bounds on q:

$$\left\lceil \frac{Mp + c_0 - c_1}{Mc_0} \right\rceil - 1 \le q \le \left\lfloor \frac{Mp + c_0 - c_1}{Mc_0} \right\rfloor - 1$$

Again, these represent values for which $send_set(p, q)$ can be nonempty. The expressions for the communication sets in Theorem 4.2 take these bounds into account.

We now turn our attention to the iteration sets. We obtain expressions for these by applying Lemma 4.4 to Equations 3.27 and 3.28.

$$\begin{aligned}
local_iter(p) &= exec(p) \cap deref(p) \\
&= [\,bot(p), top(p)\,] \cap \left[\left\lceil \frac{Mp - c_1}{c_0} \right\rceil, \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor \right] \\
&= \left[\max\left(bot(p), \left\lceil \frac{Mp - c_1}{c_0} \right\rceil \right), \min\left(top(p), \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor \right) \right] \\
nonlocal_iter(p) &= exec(p) - deref(p) \\
&= [\,bot(p), top(p)\,] - \left[\left\lceil \frac{Mp - c_1}{c_0} \right\rceil, \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor \right] \\
&= \left[bot(p), \min(top(p), \left\lceil \frac{Mp - c_1}{c_0} \right\rceil - 1) \right] \cup \\
&\qquad \left[\max\left(bot(p), \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor + 1 \right), top(p) \right]
\end{aligned}$$

The equation for $local_iter(p)$ is now in its final form, suitable for use in an implementation. The formula for $nonlocal_iter(p)$, however, is the union of two disjoint ranges, which is more difficult to use. This situation is analogous to the situation for the communication sets, in which we have a correct but computationally expensive formula. As we did then, we now identify simplifying cases; in particular, we find cases in which one of the unioned ranges is empty.

The range

$$\left[bot(p), \min(top(p), \left\lceil \frac{Mp - c_1}{c_0} \right\rceil - 1) \right]$$

in the general expression for $nonlocal_iter(p)$ will be empty if

$$bot(p) \ge \left\lceil \frac{Mp - c_1}{c_0} \right\rceil \ge \frac{Mp - c_1}{c_0}$$

Since $bot(p) \ge Mp$, a necessary condition for this is

$$Mp \ge \frac{Mp - c_1}{c_0}$$

Rewriting this expression we obtain

$$c_1 \ge Mp(1 - c_0)$$

Since M, p and c_0 are positive integers, we can deduce that this range will be empty on all processors whenever $c_1 \ge 0$. Similarly, the range

$$\left[\max(bot(p), \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor + 1), top(p) \right]$$

will be empty if

$$Mp + M - 1 \le \frac{M \cdot p + M - 1 - c_1}{c_0}$$

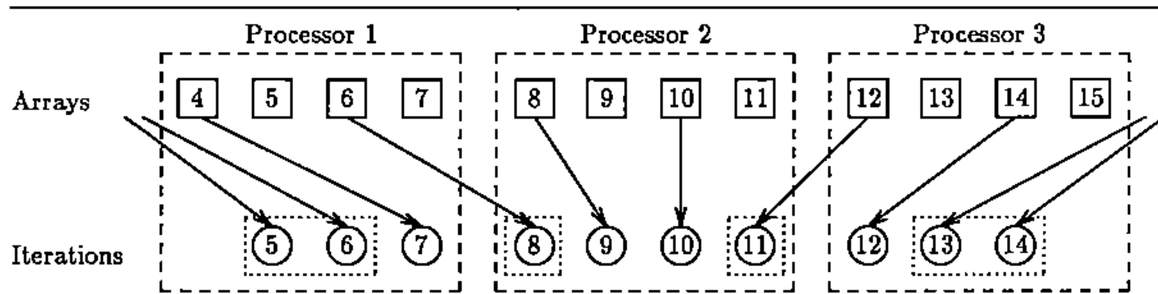


Figure 4.5: Example of *nonlocal_iter*(p) sets

or, equivalently, if

$$c_1 \leq (Mp + M - 1)(1 - c_0)$$

Again, we can bound this expression to deduce that this range will be empty on all processors if $c_1 \leq (MP - 1)(1 - c_0)$, or $c_1 \leq (N - 1)(1 - c_0)$ if N is divisible by P. These give us the conditions for Equations 4.23a and 4.23b. (There is some ambiguity as to the case to be used if $c_0 = 1$ and $c_1 = 0$; in this case, however, all three branches give $nonlocal_iter(p) = \phi$, so the expression is still consistent.) The derivation above also proves Lemma 4.5, which will be used in dealing with the special case of $c_0 = 1$.

Lemma 4.5 *If the subscripting function is* $f(i) = c_0 i + c_1$ *and* $c_0 > 0$, *then*

$$\left[bot(p), \min\left(top(p), \left\lceil \frac{Mp - c_1}{c_0} \right\rceil - 1\right)\right]$$

is empty if $c_1 \geq Mp(1 - c_0)$ *and*

$$\left[\max\left(bot(p), \left\lfloor \frac{Mp + M - 1 - c_1}{c_0} \right\rfloor + 1\right), top(p)\right]$$

is empty if $c_1 \leq (Mp + M - 1)(1 - c_0)$.

If $(N - 1)(1 - c_0) < c_1 < 0$, different processors will have different behaviors regarding whether the ranges are empty. For example, if $P \geq 4$, $M = 4$, $f(i) = 2i - 10$ (i.e. $c_0 = 2$ and $c_1 = -10$), $low = 5$, and $high = 14$. Then we have

$$\begin{aligned} nonlocal_iter(1) &= \{5\} \\ nonlocal_iter(2) &= \{8\} \cup \{11\} \\ nonlocal_iter(1) &= \{13, 14\} \end{aligned}$$

Figure 4.5 illustrates this situation, showing the *nonlocal_iter*(p) sets as dotted boxes. Note that this example shows three different behaviors on three processors. In these cases, no simplification of *nonlocal_iter*(p) is possible. □

The analysis of the communication and iteration sets for the $c_0 < 0$ case parallels the $c_0 > 0$ case. Because of the close similarity, we present only the main result here. The full proof can be found in [Koe90].

Theorem 4.3 *If all arrays in a* **forall** *are distributed by* **block**, *the subscripting function is* $f(i) = c_0 i + c_1$, *and* $c_0 < 0$, *then let*

$$\begin{aligned} lb(p,q) &= \max(c_0 top(p) + c_1, nxt(Mq, c_1, |c_0|)) \\ ub(p,q) &= \min(c_0 bot(p) + c_1, Mq + M - 1) \end{aligned}$$

Then:

If $\left\lceil \frac{c_1+1-c_0}{M} \right\rceil + c_0(p+1) - 1 \le q \le \left\lfloor \frac{c_1}{M} \right\rfloor + c_0 p$ *then*

$$recv_set(p,q) = [\, lb(p,q),\ ub(p,q);\ |c_0|\,] \tag{4.26a}$$

Otherwise,

$$recv_set(p,q) = \phi \tag{4.26b}$$

If $\left\lceil \frac{Mp+M-1+c_0-c_1}{M c_0} \right\rceil - 1 \le q \le \left\lfloor \frac{Mp-c_1}{M c_0} \right\rfloor$ *then*

$$send_set(p,q) = [\, lb(q,p),\ ub(q,p);\ |c_0|\,] \tag{4.27a}$$

Otherwise,

$$send_set(p,q) = \phi \tag{4.27b}$$

$$local_iter(p) = \begin{matrix} \left[\max\left(bot(p), \left\lceil \frac{Mp+M-1-c_1}{c_0} \right\rceil\right),\right. \\ \left.\min\left(top(p), \left\lfloor \frac{Mp-c_1}{c_0} \right\rfloor\right)\right] \end{matrix} \tag{4.28}$$

If $c_1 \le M-1$ *then*

$$nonlocal_iter(p) = \left[\max\left(bot(p), \left\lfloor \frac{Mp-c_1}{c_0} \right\rfloor + 1\right), top(p)\right] \tag{4.29a}$$

If $c_1 \ge MP(1-c_0) + c_0(1-M)$

$$nonlocal_iter(p) = \left[bot(p), \min\left(top(p), \left\lceil \frac{Mp+M-1-c_1}{c_0} \right\rceil - 1\right)\right] \tag{4.29b}$$

Otherwise,

$$nonlocal_iter(p) = \begin{matrix} \left[\max(bot(p), \left\lfloor \frac{Mp-c_1}{c_0} \right\rfloor + 1), top(p)\right] \cup \\ \left[bot(p), \min\left(top(p), \left\lceil \frac{Mp+M-1-c_1}{c_0} \right\rceil - 1\right)\right] \end{matrix} \tag{4.29c}$$

We now specialize Theorems 4.2 and 4.3 to the important cases of $|c_0| = 1$.

Theorem 4.4 *If all arrays in a* **forall** *are distributed by* **block** *and the subscripting function is* $f(i) = i + c$, *then let*

$$\begin{aligned} lb(p,q) &= \max(bot(p)+c, Mq) \\ ub(p,q) &= \min(top(p)+c, Mq+M-1) \end{aligned}$$

Then:

If $\left\lceil \frac{c+1}{M} \right\rceil + p - 1 \le q \le \left\lfloor \frac{c-1}{M} \right\rfloor + p + 1$ *then*

$$recv_set(p,q) = [\, lb(p,q), ub(p,q)\,] \tag{4.30a}$$

Otherwise,

$$recv_set(p,q) = \phi \tag{4.30b}$$

If $\lceil \frac{1-c}{M} \rceil + p - 1 \leq q \leq \lfloor \frac{-1-c}{M} \rfloor + p + 1$ *then*

$$send_set(p,q) = [\, lb(q,p), ub(q,p)\,] \tag{4.31a}$$

Otherwise,

$$send_set(p,q) = \phi \tag{4.31b}$$

$$local_iter(p) = [\,\max(bot(p), Mp - c), \min(top(p), Mp + M - 1 - c)\,] \tag{4.32}$$

If $c > 0$ *then*

$$nonlocal_iter(p) = [\,\max(bot(p), Mp + M - c), top(p)\,] \tag{4.33a}$$

If $c < 0$ *then*

$$nonlocal_iter(p) = [\, bot(p), \min(top(p), Mp - c + 1)\,] \tag{4.33b}$$

Otherwise,

$$nonlocal_iter(p) = \phi \tag{4.33c}$$

Proof. Substitute $c_0 = 1$ and $c_1 = c$ in Theorem 4.2. The bounds on $nonlocal_iter(p)$ can be improved by using Lemma 4.5 directly rather than bounding the expressions. □

Theorem 4.5 *If all arrays in a* forall *are distributed by* block *and the subscripting function is* $f(i) = c - i$, *then let*

$$\begin{aligned} lb(p,q) &= \max(c - top(p), Mq) \\ ub(p,q) &= \min(c - bot(p), Mq + M - 1) \end{aligned}$$

Then:

If $\lceil \frac{c+2}{M} \rceil - p - 2 \leq q \leq \lfloor \frac{c}{M} \rfloor - p$ *then*

$$recv_set(p,q) = [\, lb(p,q), ub(p,q)\,] \tag{4.34a}$$

Otherwise,

$$recv_set(p,q) = \phi \tag{4.34b}$$

If $\lceil \frac{c-2}{M} \rceil - p - 2 \leq q \leq \lfloor \frac{c}{M} \rfloor - p$ *then*

$$send_set(p,q) = [\, lb(q,p), ub(q,p)\,] \tag{4.35a}$$

Otherwise,

$$send_set(p,q) = \phi \tag{4.35b}$$

$$local_iter(p) = [\,\max(bot(p), c + 1 - Mp - M), \min(top(p), c - Mp)\,] \tag{4.36}$$

```
-- Code on processor p

const
   -- user defined constants omitted
   C = N / NP;
   -- first processor to send to and bounds on send_set
   q1 = p - 1 + ceil( -16 / C );
   low_send_q1 = max( low+17, C*p, C*q1+17 );
   high_send_q1 = min( high+17, C*(p+1)-1, C*(q1+1)+16 );
   -- second processor to send to and bounds on send_set
   q2 = p + floor( (C-18) / C );
   low_send_q2 = max( low+17, C*p, C*q2+17 );
   high_send_q2 = min( high+17, C*(p+1)-1, C*(q2+1)+16 );
   -- first processor to receive from and bounds on recv_set
   q3 = p - 1 + ceil( 18 / C );
   low_recv_q3 = max( low+17, C*q3, C*p+17 );
   high_recv_q3 = min( high+17, C*(q3+1)-1, C*(p+1)+16 );
   -- second processor to receive from and bounds on recv_set
   q4 = p + 1 + floor( 16 / C );
   low_recv_q4 = max( low+17, C*q4, C*p+17 );
   high_recv_q4 = min( high+17, C*(q4+1)-1, C*(p+1)+16 );
   -- bounds on local_iter(p)
   low_local = max( low, C*p );
   high_local = min( high, C*p+C-18 );
   -- bounds on nonlocal_iter(p)
   low_nonlocal = max( low, C*p, C*p-17 );
   high_nonlocal = min( high, C*p+C-1 );

var
   A, B : array[ 0..N-1 ] of real dist by [ block ] on procs;
   temp : array[ min(low_recv_q3,low_recv_q4) .. max(high_recv_q3,high_recv_q4) ]
         of real;              -- compiler-generated buffer

-- communication statements: sending
if ( q1<>p and low_send_q1<=high_send_q1 ) then
   send( B[ low_send_q1..high_send_q1 ], procs[q1] );
end;
if ( q2<>p and q2<>q1 and low_send_q2<=high_send_q2 ) then
   send( B[ low_send_q2..high_send_q2 ], procs[q2] );
end;
-- local computations
for i in low_local..high_local do
   A[ i ] := B[ i + 17 ];
end;
-- communication statements: receiving
if ( q3<>p and low_send_q3<=high_send_q3 ) then
   temp[ low_recv_q3..high_recv_q3 ] := recv( procs[q3] );
end;
if ( q4<>p and low_recv_q4<=high_recv_q4 ) then
   temp[ low_recv_q4..high_recv_q4 ] := recv( procs[q4] );
end;
-- nonlocal computations
for i in low_nonlocal..high_nonlocal do
   A[ i ] := temp[ i + 17 ];
end;
```

Figure 4.6: Implementation of Figure 4.4 when $c_0 = 1$, $c_1 = 17$

```
-- Code on processor p

const
   -- user-defined constants omitted
   C = N / NP;
   -- bounds on processors to send to
   low_send_p = ceil( -p/2 - (C-19)/(2*C) );
   high_send_p = floor( -p/2 + 8/C );
   -- bounds on processors to receive from
   low_recv_p = -2*p - 3 + ceil( 19 / C );
   high_recv_p = -2*p + floor( 16 / C );
   -- bounds on local iterations
   low_local = max( low, C*p, ceil( (C*p+C-17) / -2 ) );
   high_local = min( high, C*p+C-1, floor( (C*p-16) / -2 ) );
   -- bounds on nonlocal iterations
   low_nonlocal_1 = max( low, C*p );
   high_nonlocal_1 = min( high, C*p+C-1, ceil( (C*p+C-17) / -2 ) );
   low_nonlocal_2 = max( low, C*p, floor( (C*p-16) / -2 ) );
   high_nonlocal_2 = min( high, C*p+C-1 );
   -- bounds on temporary array
   low_temp = 18 - 2*C*p - 2*C;
   high_temp = 16 - 2*C*p;;

var
   A, B : array[ 0..N-1 ] of real dist by [ block ] on procs;
   temp : array[ low_temp..high_temp ] of real;              -- compiler temporary

-- communications statements: sending
for q in low_send_p..high_send_p do
   low_bound := max( 16-2*high, 18-2*C*q-2*C, C*p+(C*p)%2 );
   high_bound := min( 16-2*low, 16-2*C*q, C*p+C-1 );
   if ( q<>p and low_bound<=high_bound ) then
         send( B[ low_bound..high_bound by 2 ], procs[q] );
   end;
end;
-- local computations
for i in low_local..high_local do
   A[ i ] := B[ 16 - 2*i ];
end;
-- communications statements: receiving
for q in low_recv_p..high_recv_p do
   low_bound := max( 16-2*high, 14-2*C*p-2*C, C*q+(C*q)%2 );
   high_bound := min( 16-2*low, 16-2*C*p, C*q+C-1 );
   if ( q<>p and low_bound<=high_bound ) then
         temp[ low_bound..high_bound by 2 ] := send( procs[q] );
   end;
end
-- nonlocal computations
for i in low_exec..low_local-1 do
   A[ i ] := temp[ 16 - 2 * i ];
end;
for i in high_local+1..high_exec do
   A[ i ] := temp[ 16 - 2 * i ];
end;
```

Figure 4.7: Implementation of Figure 4.4 when $c_0 = -2$, $c_1 = 16$

If $c \leq M - 1$ *then*

$$nonlocal_iter(p) = [\max(bot(p), c + 1 - Mp), top(p)] \tag{4.37a}$$

If $c \geq 2N + M - 1$ *then*

$$nonlocal_iter(p) = [bot(p), \min(top(p), c - Mp - M)] \tag{4.37b}$$

Otherwise

$$nonlocal_iter(p) = \begin{array}{c}[\max(bot(p), c - Mp + 1), top(p)] \cup \\ [bot(p), \min(top(p), c - Mp - M)]\end{array} \tag{4.37c}$$

Proof. Substitute $c_0 = -1$ and $c_1 = c$ into Theorem 4.3. No simplification of $nonlocal_iter(p)$ is possible in this case. □

Figures 4.6 and 4.7 show how the formulas derived in this section can be used directly in an implementation. Ranges of iterations correspond directly to for loop bounds and steps, while ranges of array subscripts describe sections of arrays. One issue of some subtlety is allocation of memory to hold off-processor data. The size of the necessary buffers must be calculated from the cardinality of the sets; given this information, temporary variables for the buffers can be statically declared or dynamically allocated. (The current Kali implementation uses dynamic allocation.) To avoid sparse use of storage, addressing into these buffers should ensure that the range is stored contiguously. This forces the range step size to be accounted for in the addressing formulas. For illustration here, we will assume that the implementation language allows arrays to be declared with non-unit step sizes in the subscript ranges; in a native code implementation, a division by the step size would be required in the address calculation. Some additional optimizations are possible if $|c_0| = 1$. Buffer addressing need not be complex, since the ranges for the communication sets have unit stride. Also, under the assumptions we have made in this section, the conditions on q in the communication sets allow at most two sets to be nonempty. These values of q can be kept explicitly for quick reference. Using these ideas, Figure 4.6 shows an implementation of Figure 4.4 with the subscripting function $f(i) = i + 17$. Figure 4.7 does the same for $f(i) = 16 - 2i$.

4.4 Cyclic Distributions with Linear Subscripts

In this section we consider programs like the one shown in Figure 4.8. The conditions in this section are identical to those in Section 4.3, except that all array distributions will be cyclic rather than block. In particular, we again assume a linear subscripting function $f(i) = c_0 i + c_1$ where $c_0 \neq 0$. A practical example of such a program would be a cyclic reduction tridiagonal system solver. (The partial differential equation solvers mentioned in the last section could also be implemented with cyclic data distributions, but usually are not because of the increased communication requirements.) Theorem 4.6 gives the basic results for the case of $c_0 > 0$, while Theorem 4.7 handles the $c_0 < 0$ case. Again, the special cases of $|c_0| = 1$ are of interest; we therefore derive forms for these cases in Theorem 4.8 and Theorem 4.9.

Once again we will need expressions for $local(p)$ and $exec(p)$, which were derived in Chapter 3. We give those equations here for convenience.

$$local(p) = \{ i \mid i \equiv p \pmod{P} \} \tag{4.38}$$

$$\begin{aligned} exec(p) &= [low, high] \cap local(p) \\ &= [nxt(low, p, P),\ high;\ P] \end{aligned} \tag{4.39}$$

As in the last section, the bounds on $exec(p)$ will be useful. Thus, we make the following definition.

```
processors procs : array[ 0..P-1 ] with P in 1..max_procs;

const
   N = ...;
   low = ...;
   high = ...;
   c0 = ...;
   c1 = ...;

var
   A, B : array[ 0..N-1 ] of real dist by [ cyclic ] on procs;

forall i in low..high on A[i].loc do
   A[ i ] := B[ c0*i + c1 ];
end;
```

Figure 4.8: Program using linear subscript functions and cyclic array distributions

Definition 4.5 *Define $bot(p)$ and $top(p)$ as*

$$bot(p) = nxt(low, p, P) \tag{4.40}$$

$$top(p) = nxt(high - P + 1, p, P) \tag{4.41}$$

The definition makes $top(p)$ the largest integer less than $high$ which is equivalent to p modulo P. This is the exact upper bound on the range, rather than an upper bound which is not reached.

To derive expressions for the communication and iteration sets when $c_0 > 0$ we will need the following lemma.

Lemma 4.6 *Let the subscripting function be $f(i) = c_0 i + c_1$, and $c_0 > 0$, let $G = \gcd(P, c_0)$ and let n and m be such that $c_0 n + Pm = \gcd(P, c_0)$. If all arrays in a* **forall** *are distributed by* cyclic, *then*

$$ref(p) = [c_0 bot(p) + c_1,\ c_0 top(p) + c_1;\ c_0 P] \tag{4.42}$$

$$deref(p) = \begin{cases} \phi & \text{if } p \not\equiv c_1 \pmod{G} \\ \{ i \mid i \equiv \frac{n(p-c_1)}{G} \pmod{P/G} \} & \text{if } p \equiv c_1 \pmod{G} \end{cases} \tag{4.43}$$

Proof.

$$\begin{aligned} ref(p) &= f(exec(p)) \\ &= \{ c_0 i + c_1 \mid i \equiv p \pmod{P} \text{ and } bot(p) \le i \le top(p) \} \\ &= \{ c_0 i + c_1 \mid c_0 i + c_1 \equiv c_0 p + c_1 \pmod{c_0 P} \text{ and } \\ &\qquad c_0 bot(p) + c_1 \le c_0 i + c_1 \le c_0 top(p) + c_1 \} \\ &= [c_0 bot(p) + c_1,\ c_0 top(p) + c_1;\ c_0 P] \end{aligned}$$

(The last step is valid because $bot(p) \equiv p \pmod{P}$, which implies $c_0 bot(p) + c_1 \equiv c_0 p + c_1 \pmod{c_0 P}$.) This proves Equation 4.42

$$\begin{aligned} deref(p) &= f^{-1}(local(p)) \\ &= \{ i \mid c_0 i + c_1 \equiv p \pmod{P} \} \end{aligned}$$

By reasoning similar to that in Lemma 4.2, if $c_1 \not\equiv p \pmod{\gcd(P, c_0)}$ then the last set is empty, and if $c_1 \equiv p \pmod{\gcd(P, c_0)}$ then all elements of $deref(p)$ are equivalent modulo $P/\gcd(P, c_0)$.

To find a solution, we use the values of n and m returned by the Extended Euclid's algorithm [AHU74] for which $c_0n+Pm = \gcd(P, c_0)$. These give $c_0 n \frac{p-c_1}{\gcd(P,c_0)} = -Pm\frac{p-c_1}{\gcd(P,c_0)} \equiv p \pmod{P/\gcd(P, c_0)}$. This proves Equation 4.43. □

We can now prove Theorem 4.6.

Theorem 4.6 *If all arrays in a* **forall** *are distributed by* **cyclic**, *the subscripting function is* $f(i) = c_0 i + c_1$ *and* $c_0 > 0$, *then let* $G = \gcd(P, c_0)$ *and let* n *and* m *be such that* $c_0 n + Pm = \gcd(P, c_0)$. *Then:*

If $q \equiv c_0 p + c_1 \pmod{P}$ *then*

$$recv_set(p,q) = [c_0 bot(p) + c_1,\ c_0 top(p) + c_1;\ c_0 P] \tag{4.44a}$$

Otherwise,

$$recv_set(p,q) = \phi \tag{4.44b}$$

If $p \equiv c_1 \pmod{G}$ *and* $q \in \left[\left(\frac{n(p-c_1)}{G}\right) \% \left(\frac{P}{G}\right),\ P-1;\ \frac{P}{G}\right]$ *then*

$$send_set(p,q) = [c_0 bot(q) + c_1,\ c_0 top(q) + c_1;\ c_0 P] \tag{4.45a}$$

Otherwise,

$$send_set(p,q) = \phi \tag{4.45b}$$

If $p \equiv c_1 \pmod{G}$ *and* $p \equiv \frac{n(p-c_1)}{G} \pmod{P/G}$ *then*

$$local_iter(p) = exec(p) \tag{4.46a}$$

Otherwise,

$$local_iter(p) = \phi \tag{4.46b}$$

If $p \not\equiv c_1 \pmod{G}$ *or* $p \not\equiv \frac{n(p-c_1)}{G} \pmod{P/G}$ *then*

$$nonlocal_iter(p) = exec(p) \tag{4.47a}$$

Otherwise,

$$nonlocal_iter(p) = \phi \tag{4.47b}$$

Proof.

We obtain the communication set by applying Lemma 4.6 to Equation 3.20.

$$\begin{aligned} recv_set(p,q) &= local(q) \cap ref(p) \\ &= \{\, i \mid i \equiv q \pmod{P} \,\} \cap [c_0 bot(p) + c_1,\ c_0 top(p) + c_1;\ c_0 P] \\ &= \begin{cases} [c_0 bot(p) + c_1,\ c_0 top(p) + c_1;\ c_0 P] & \text{if } c_0 bot(p) + c_1 \equiv q \pmod{P} \\ \phi & \text{otherwise} \end{cases} \\ &= \begin{cases} [c_0 bot(p) + c_1,\ c_0 top(p) + c_1;\ c_0 P] & \text{if } c_0 p + c_1 \equiv q \pmod{P} \\ \phi & \text{otherwise} \end{cases} \end{aligned}$$

(The last step follows because $bot(p) \equiv p \pmod{P}$.) This form is efficiently computable, since for each p there will be exactly one q with a nonempty $recv_set(p,q)$, easily found by taking the remainder of $c_0p + c_1$. The corresponding expression for $send_set(p,q)$,

$$send_set(p,q) = \begin{cases} [c_0 bot(q) + c_1,\ c_0 top(q) + c_1;\ c_0 P] & \text{if } c_0 q + c_1 \equiv p \pmod{P} \\ \phi & \text{otherwise} \end{cases}$$

is not acceptable, because there may be a number of nonempty sets which cannot be quickly computed from this form. We therefore characterize the solutions to $c_0q + c_1 \equiv p \pmod{P}$. The reasoning used to derive Equation 4.43 shows that

$$q \equiv \frac{n(p - c_1)}{\gcd(P, c_0)} \pmod{P/\gcd(P, c_0)}$$

To this we add that processor numbers are in the range $[0, P-1]$ to derive an explicit condition on q.

$$q \in \left[\left(\frac{n(p - c_1)}{\gcd(P, c_0)}\right) \% \left(\frac{P}{\gcd(P, c_0)}\right),\ P - 1;\ \frac{P}{\gcd(P, c_0)} \right]$$

where n is chosen so that $c_0n + Pm = \gcd(P, c_0)$ by the Extended Euclid's algorithm. As q is now defined as a member of a range, it is possible to iterate over that range producing all legal values of q. We thus have the required formula for $send_set(p,q)$.

To derive the iteration sets, we apply Equation 4.43 to Equations 3.27 and 3.28.

$$\begin{aligned} local_iter(p) &= exec(p) \cap deref(p) \\ &= [\,bot(p),\ top(p);\ P\,] \cap deref(p) \end{aligned}$$

If $p \not\equiv c_1 \pmod{\gcd(P, c_0)}$, the intersection is clearly empty. Otherwise, if

$$bot(p) \not\equiv \frac{n(p - c_1)}{\gcd(P, c_0)} \pmod{P/\gcd(P, c_0)}$$

then the intersection is again empty. This leaves the case of

$$bot(p) \equiv \frac{n(p - c_1)}{\gcd(P, c_0)} \pmod{P/\gcd(P, c_0)}$$

In this case, since $\frac{P}{\gcd(P,c_0)}$ is a factor of P, all elements of $exec(p)$ will be in $deref(p)$. Thus, we have shown that if

$$bot(p) \equiv c_1 \pmod{\gcd(P, c_0)} \wedge bot(p) \equiv \frac{n(p - c_1)}{\gcd(P, c_0)} \pmod{P/\gcd(P, c_0)}$$

then $local_iter(p) = [\,bot(p),\ top(p);\ P\,] = exec(p)$, and that the set is null otherwise. Since $bot(p) \equiv p \pmod{P}$, p can be substituted for $bot(p)$ in the above condition to give Equation 4.46. The expression for $nonlocal_iter(p)$ can be derived immediately from this equation by noting that the sets are complements. □

The corresponding analysis for the $c_0 < 0$ case is given by Theorem 4.7. We omit the proof here because it is so similar to that of Theorem 4.6; the complete proof can be found in [Koe90].

Theorem 4.7 *If all arrays in a* **forall** *are distributed by* **cyclic**, *the subscripting function is* $f(i) = c_0 i + c_1$, *and* $c_0 < 0$, *then let* $G = \gcd(P, c_0)$ *and let* n *and* m *be such that* $c_0n + Pm = \gcd(P, c_0)$. *Then:*

If $q \equiv c_0 p + c_1 \pmod{P}$ *then*

$$recv_set(p,q) = [\, c_0 top(p) + c_1,\ c_0 bot(p) + c_1;\ |c_0|P\,] \tag{4.48a}$$

Otherwise,

$$recv_set(p,q) = \phi \tag{4.48b}$$

If $p \equiv c_1 \pmod{G}$ *and* $q \in \left[\, \left(\frac{n(p-c_1)}{G}\right) \% \left(\frac{P}{G}\right),\ P-1;\ \frac{P}{G} \right]$ *then*

$$send_set(p,q) = [\, c_0 top(q) + c_1,\ c_0 bot(q) + c_1;\ |c_0|P\,] \tag{4.49a}$$

Otherwise,

$$send_set(p,q) = \phi \tag{4.49b}$$

If $p \equiv c_1 \pmod{\gcd(P, c_0)}$ *and* $p \equiv \frac{n(p-c_1)}{G} \pmod{P/G}$ *then*

$$local_iter(p) = exec(p) \tag{4.50a}$$

Otherwise,

$$local_iter(p) = \phi \tag{4.50b}$$

If $p \not\equiv c_1 \pmod{\gcd(P, c_0)}$ *or* $p \not\equiv \frac{n(p-c_1)}{G} \pmod{P/G}$ *then*

$$nonlocal_iter(p) = exec(p) \tag{4.51a}$$

Otherwise,

$$nonlocal_iter(p) = \phi \tag{4.51b}$$

We now specialize Theorems 4.6 and 4.7 to the cases of $c_0 = 1$ and $c_0 = -1$.

Theorem 4.8 *If all arrays in a* forall *are distributed by* cyclic*and the subscripting function is* $f(i) = i + c$, *then:*

If $q = (p + c) \% P$ *then*

$$recv_set(p,q) = [\, bot(p) + c,\ top(p) + c;\ P\,] \tag{4.52a}$$

Otherwise,

$$recv_set(p,q) = \phi \tag{4.52b}$$

If $q = (p - c) \% P$ *then*

$$send_set(p,q) = [\, bot(q) + c,\ top(q) + c;\ c_0 P\,] \tag{4.53a}$$

Otherwise,

$$send_set(p,q) = \phi \tag{4.53b}$$

If $c \,\%\, P = 0$ *then*

$$local_iter(p) = exec(p) \tag{4.54a}$$

Otherwise,

$$local_iter(p) = \phi \tag{4.54b}$$

If $c \,\%\, P \neq 0$ *then*

$$nonlocal_iter(p) = exec(p) \tag{4.55a}$$

Otherwise,

$$nonlocal_iter(p) = \phi \tag{4.55b}$$

Theorem 4.9 *If all arrays in a* **forall** *are distributed by* **cyclic**, *the subscripting function is* $f(i) = c - i$, *then let* n *and* m *be such that* $c_0 n + Pm = \gcd(P, c_0)$. *Then:*

If $q = (c - p) \,\%\, P$ *then*

$$recv_set(p, q) = [\, c - top(p),\ c - bot(p);\ P\,] \tag{4.56a}$$

Otherwise,

$$recv_set(p, q) = \phi \tag{4.56b}$$

If $q = (c - p) \,\%\, P$ *then*

$$send_set(p, q) = [\, c - top(q),\ c - bot(q);\ P\,] \tag{4.57a}$$

Otherwise,

$$send_set(p, q) = \phi \tag{4.57b}$$

If $p = (c - p) \,\%\, P$ *then*

$$local_iter(p) = exec(p) \tag{4.58a}$$

Otherwise,

$$local_iter(p) = \phi \tag{4.58b}$$

If $p \neq (c - p) \,\%\, P$ *then*

$$nonlocal_iter(p) = exec(p) \tag{4.59a}$$

Otherwise,

$$nonlocal_iter(p) = \phi \tag{4.59b}$$

The proofs follow by simple substitution.

The remarks at the end of Section 4.3 regarding the use of theorems in implementation apply here as well. We demonstrate this by transforming Figure 4.8 using the same subscripting functions as in that section. Figure 4.9 implements $f(i) = i + 17$, assuming P is not divisible by 17. Figure 4.10 implements $f(i) = 16 - 2i$, making no assumptions about P.

```
-- Code on processor p
const
   -- user constants omitted
   low_exec = low + (p-low)%P;            -- bounds on exec(p)
   high_exec = high-P+1 + (p-high+1)%P;
   q_recv = (p+17) % P;            -- recv_set parameters
   low_recv = low_exec + 17;
   high_recv = high_exec + 17;
   q_send = (p-17) % P;            -- send_set parameters
   low_send = low + (q_send-low)%P + 17;
   high_send = high-P+1 + (q_send-high+1)%P + 17;
var
   A, B : array[ 0..N-1 ] of real dist by [ cyclic ] on procs;
   temp : array[ p..N-1 by P ] of real;            -- compiler temporary

-- communication statements: sending
if ( low_send<=high_send ) then
   send( B[ low_send .. high_send by P ], procs[q_send] );
end;
-- no local computations
-- communication statements: receiving
if ( low_recv<=high_recv ) then
   temp[ low_recv .. high_recv by P ] := recv( procs[q_recv] );
end;
-- nonlocal computations
for i in low_exec..high_exec do
   A[ i ] := temp[ i + 17 ];
end;
```

Figure 4.9: Implementation of Figure 4.8 when $c_0 = 1$, $c_1 = 17$.

```
-- Code on processor p

const
   NP = NP1 - 1;
   low = ...;
   high = ...;
   -- bounds on exec
   low_exec = low + (p-low)%NP;
   high_exec = high-NP+1 + (p-high+1)%NP;
   -- gcd(P,c0), n, m
   g, n, m = extended_euclid( NP, c0 );
   p_over_gcd = NP / g;
   -- bounds on recv_set
   q_recv = (16-2*p) % P;
   low_recv = 16 - 2 * high_exec;
   high_recv = 16 - 2 * low_exec;
   step_recv = 2 * NP;
   -- constants for send_set
   q_send = ((n*(p-16))/g) % p_over_gcd;

var
   A, B : array[ 0..N-1 ] of real dist by [ cyclic ] on procs;
   temp : array[ low_recv .. high_recv by step_recv ] of real;        -- compiler temporaries
   low_send, high_send : integer;

-- communication statements: sending
if ( p%g = 16%g ) then
   for q in q_send..P-1 by p_over_gcd do
         low_send := 16 - 2 * (high-P+1+(q-high+1)%NP);
         high_send := 16 - 2 * (low+(q-low)%NP);
         if ( p<>q and low_send<=high_send ) then
            send( B[ low_send .. high_send by step_recv ], procs[q] );
         end;
   end;
end;
-- local computations
if ( p%g=16%g and p%p_over_g=q_send ) then
   for i in low_exec..high_exec do
         A[ i ] := B[ 16 - 2*i ];
   end;
end;
-- communication statements: receiving
if ( q_recv<>p and low_recv<=high_recv ) then
   temp[ low_recv .. high_recv by step_recv ] := recv( procs[q_recv] );
end;
-- nonlocal computations
if ( p%g<>16%g or p%p_over_g<>q_send ) then
   for i in low_exec..high_exec do
         A[ i ] := B[ 16 - 2*i ];
   end;
end;
```

Figure 4.10: Implementation of Figure 4.8 when $c_0 = -2$, $c_1 = 16$

4.5 Extensions to Compile-time Analysis

Many extensions to the above analysis are possible. This section only mentions extensions likely to be useful in practice.

Block-cyclic distribution patterns can be analyzed by techniques similar to those in Sections 4.3 and 4.4. For the subscript functions $f(i) = i + c$ and $f(i) = c - i$ the communication and iteration sets are unions of sets of the forms shown in Theorems 4.4 and 4.5. The unions essentially have one set for each block stored on the processor. Additional conditions checking for empty sets can also be found. It is harder to obtain closed expressions for general linear subscripts because of the added complexity of the *local* function. The principle of a union with one set for each block still applies, but now each block may produce another union. We will report more fully on these distributions in the future. Other distribution patterns, such as skewed distribution, also appear amenable to compile-time analysis. Because of the wide variety of distributions, the detail of the analysis needed, and the tediousness of the symbol manipulation, automation of this analysis is attractive. Section 7.3 considers this possibility.

Multiple-dimensional arrays may be analyzed in the same way as one-dimensional arrays if two conditions hold.

1. The multidimensional distribution patterns are combinations of one-dimensional patterns, such as the patterns given by Equations 3.8 through 3.11.

2. The subscripts are separable, that is, the same **forall** index does not appear in more than one dimension.

In these cases, the multidimensional communication and iteration sets are Cartesian products of the corresponding sets for each individual dimension. If fundamentally multidimensional distributions (such as the skewed distribution of Equation 3.13) are used, then new theorems must be proved to describe the sets. Likewise, coupled subscripts require new methods of analysis.

If several subscripted references are made in a **forall**, it is possible that the iteration sets for different references will differ. In this case, the set of purely local iterations will be the intersection of the $local_iter(p)$ sets for each reference, which can be calculated from Lemma 4.2 if necessary. All references in these iterations can be satisfied from the local arrays, so implementation is straightforward. Similarly, the set of nonlocal iterations is the union of the individual $nonlocal_iter(p)$ sets. While this may not be a range, it is possible to implement the iteration by means of either nested or repeated loops. Implementation of the nonlocal iterations, which may satisfy some references from local arrays and some from temporary message buffers, is not so easy in general. In some special cases, all of the sets will be the same for all of the subscripts; in particular, this is true when all of the subscript functions are the same. In this case no additional implementation steps are necessary. In other cases, a mechanism is needed for correctly accessing the correct data structure (local array or message buffer) in the nonlocal iterations.

Several approaches to implementing nonlocal iterations in the presence of multiple subscripts are possible.

1. Check the locality of each reference, and satisfy the reference from the local array or the communication buffer as appropriate.

2. Generate a loop for every possible combination of local and nonlocal references. Within each loop, no testing is needed for access.

3. Store the nonlocal array references as an overlap region around the original array. All references then access the local array.

4. Copy elements of the local array into an overdimensioned message buffer. All references then access the buffer.

The current Kali compiler implements the first approach because of its conceptual simplicity. This approach has the disadvantage of slow execution speed because of the overhead of locality tests. Generating multiple loops produces fast code, but the number of loops may grow exponentially. The last two approaches avoid the disadvantages mentioned above, but raise the problem of calculating the size of the overlap or buffer; Gerndt [Ger89] has produced some interesting results in this area, but not all cases can be handled efficiently yet. The last alternative also has the disadvantage of copying overhead.

We have already stated that our results apply to expressions that are invariant for all iterations of the **forall**. The theorems in Sections 4.2, 4.3, and 4.4 can be used without change for invariants, but it is worth noting how the code generation must change. All calculations using run-time invariants rather than compile-time constants must be performed at run time. These calculations must also be repeated whenever the invariant value may have changed; in the worst case, this implies recalculation on every execution of the **forall**. This may result in complex code if different cases must be generated, as when the compiler cannot deduce the sign of the **forall** index's coefficient. In that case, a conditional must be used with branches for positive and negative values. Because only integer operations will be required in any case, the resulting code will still be quite efficient. Other cases, including Gaussian elimination and cyclic reduction, do not need such branching because deductions can easily be made about coefficient values.

Despite the effort that can be put into developing new theorems for compile-time analysis, undecidability results guarantee that there will always be some subscript function that the compiler will be unable to analyze. For these cases, we must fall back on other techniques such as those shown in the next chapter.

Chapter 5

Run-time Analysis

In this chapter we consider the implementation of **forall** statements for which the compile-time analysis of Chapter 4 is not applicable. Such a loop occurs whenever either the distribution or the subscript functions cannot be computed by the compiler. In practice, these loops often arise from solving irregular problems such as relaxation on unstructured meshes or sparse linear algebra. No matter how general the compile-time analysis becomes, undecidability guarantees that there will be some cases which the compiler cannot analyze fully. In these cases the only option is to generate code which determines the communication and iteration sets at run-time. The techniques presented here for doing this have previously appeared in shorter form in [KMV90, KMSB90]. Section 5.1 introduces the *inspector-executor* strategy which forms the basis of the generated code. Section 5.2 describes the major data structures used in our current implementation of this strategy, as well as an overview of some alternatives. Section 5.3 demonstrates the inspector-executor technique on a simple but realistic example. Finally, Section 5.4 describes some additional optimizations that can be applied to the basic strategy.

5.1 The Inspector-Executor Strategy

For run-time analysis, we divide the generated code into two parts: the *inspector* and the *executor*. The inspector is responsible for "inspecting" the **forall** and generating the necessary communication and iteration sets. Briefly, this is done by executing a modified copy of the original **forall**; the exact mechanism is described in Section 5.1.1. An important consideration in implementing the inspector is that communication patterns are often used repeatedly. This observation can be exploited by only running the inspector once and saving the results. Section 5.1.1 also explains how this is done. The executor's task is to perform the actual computation of the **forall** statement, including any necessary communications. To do this, the executor relies on the sets generated by the inspector. Section 5.1.2 describes the executor.

5.1.1 The Inspector

The purpose of the inspector is to generate the communication and iteration sets needed to execute the computation. Thus, the inspector corresponds to the first step of the outline of Figure 3.2. This is most efficiently done by having each processor compute only its own sets, that is, processor p computes $local_iter(p)$, $nonlocal_iter(p)$, and $send_set(p,q)$ and $recv_set(p,q)$ for all q. These sets are stored explicitly on the processor; the other sets defined in Chapter 3 are either stored implicitly or discarded after they are used.

As Figure 5.1 shows, the inspector consists of three phases: an initialization phase, a parallel loop, and a global communication phase. The initializations simply set the processor's communication and iteration sets be be empty. The loop on processor p executes all iterations in $exec(p)$, examining

Code executed on processor p:

- Set all communication and iteration sets to empty.
- Execute a modified copy of the original **forall** in parallel.
  - Check all array references which might be made by an iteration. If a reference accesses an element on processor q, add the reference to $recv_set(p, q)$.
  - If all references in an iteration are local, add it to $local_iter(p)$; otherwise, add it to $nonlocal_iter(p)$.
- Transpose $recv_set(p, q)$ into $send_set(q, p)$ using a global communication phase.

Figure 5.1: Outline of the inspector

every array reference that might access nonlocal data and checking the locality of the actual reference. Nonlocal references are added to $recv_set(p, q)$, where q is the processor on which the reference is stored. Generating the array references amounts to a brute-force calculation of $ref(p)$; checking their locality corresponds to taking the intersection $ref(p) \cap \bigcup_{q \in P} local(q)$. While each iteration is being processed, it is simple to keep track of whether any nonlocal array accesses were actually made in that iteration and to add it to the appropriate iteration set ($local_iter(p)$ or $nonlocal_iter(p)$). This computes the iteration sets directly, rather than generating them through $deref(p)$. At the end of this loop, both iteration sets and all of the $recv_set(p, q)$ sets have been calculated. The global communication phase can then generate the $send_set(p, q)$ sets using the transposition property of communication sets ($send_set(p, q) = recv_set(q, p)$). Several algorithms exist for doing this transposition; we describe the one used in our current implementation in Section 5.2. The global communication phase completes the inspector.

The inner structure of the main inspection loop deserves further comment. A simple, but correct, implementation would duplicate the entire computation of the loop, making assignments to temporary variables to avoid overwriting live data. A more efficient version, which is used in our implementation, only duplicates the code necessary to calculate each subscript. For each subscript, this body of code is known as the "slice" of the program with respect to the variables in that expression [Wei84]. Essentially what is needed is the transitive closure of the data dependence graph at that point of the program. Examples of statements which are excluded from the inspector by using these slices are

1. Assignments to variables not used in subscripts
2. Nested for loops whose index variables are not used by subscripts

The treatment of conditionals also warrants mention. As described in Section 3.6, ignoring conditional statements which may alter the control flow produces a conservative approximation of the communication sets. Using this approximation further reduces the amount of computation in the inspector.

An important consideration in implementing the inspector is the data structures used to implement the communication and iteration sets. Because the data structures will also be used by the executor, they must be designed carefully to allow efficiency in the resulting implementation. We therefore defer discussion of these data structures to Section 5.2 and only list the operations that must be supported for the executor here. The iteration set data structure must support insertions of unique elements, while the communication set data structure must support insertion of nonunique elements and the transposition needed to compute $send_set(p, q)$.

The above description of the inspector is complete for the common case in which all of the subscripts in a **forall** can be calculated using only local data. In general, however, complex subscripts

Code on processor p:

- For all q with $send_set(p,q) \neq \phi$, send message containing $send_set(p,q)$ to q.
- Execute **forall** iterations in $local_iter(p)$, using original loop body. References to distributed arrays need no special treatment.
- For all q with $recv_set(p,q) \neq \phi$, receive message with $recv_set(p,q)$ from q and store in message buffer.
- Execute **forall** iterations in $nonlocal_iter(p)$, using modified loop body. References to distributed arrays require an access of the message buffer.

Figure 5.2: Outline of executor

may access nonlocal elements of distributed arrays (for example, if all three arrays in the expression $A[B[C[i]]]$ are distributed, then a nonlocal value from B may be needed to calculate the subscript of A). In these cases, it is necessary to have a multi-phase inspector. All subscript expressions are placed in a graph, and a directed edge is entered between two expressions if one is a component of the other. A topological sort of the expressions then produces a set of phases in which each phase depends only on the nonlocal data examined in previous phases. The phases can then be inspected one at a time, and the nonlocal data found in each phase fetched before the next phase. This scheme correctly handles arbitrary levels of indirection.

An important optimization can be made to the inspector based on the observation that in many programs the communication pattern, while unpredictable at compile-time, is static. An excellent example of this is relaxation on unstructured meshes. In these algorithms, the mesh used in the calculation is computed at the beginning of the program based on problem-specific factors. Generally these factors include the input to the problem, so the compiler cannot analyze the communication induced by the mesh directly; instead, an inspector-executor strategy is needed. The relaxation algorithm itself sweeps over the same mesh many times, thus repeating the same set of sends and receives. An efficient implementation of the relaxation can take advantage of this situation by computing the communication and iteration sets only once. Thus, the inspector is executed once, and the results used many times. This amortizes the cost of the inspector over the cost of the entire computation, rather than incurring the inspector overhead on every relaxation sweep. This optimization is used in the code generated by the Kali compiler, and appears in the code in Section 5.3 as the test of the variable $first_time$.

The above scheme for amortizing the inspector cost has several variants that are important in practice. Some unstructured mesh algorithms, such as free Laplacian algorithms, require updating the mesh periodically and continuing the computation with a new mesh. In this case the inspector must be executed after every update of the mesh. Other algorithms, such as multigrid methods, alternate between computations on a series of meshes. In these cases, several groups of communication and iteration sets will be active simultaneously. An efficient implementation must identify this situation and save the sets associated with each mesh. Neither of these generalizations has been incorporated in the present Kali compiler.

5.1.2 The Executor

The purpose of the executor is to perform the actual computation of the **forall** given the necessary communication and iteration sets. As Figure 5.2 shows, this is done by a direct implementation of the last four steps of Figure 3.2. The major issues left unresolved in Figure 5.2 are the data structures for the iteration and communication sets, which will be discussed in Section 5.2. In this section we discuss the implications of the executor outline on the design of those data structures.

The major requirement added to the iteration set data structures by the executor is the necessity

of extracting all elements in the set. This data structure traversal is necessary to control the execution of the loops in the second and fourth steps of Figure 5.2. It is not necessary to retrieve the iterations in order, however, since the semantics of **foralls** allow arbitrary execution orders.

Two new requirements are placed on the communication set data structures by the executor. First, it must be possible to extract all elements of a set to implement the communication statements (the first and third steps in the outline). The order in which elements is accessed is not vital, except that the sending and receiving processors of any message must traverse the data structures in the same order. That is, it is immaterial whether the elements appear sorted in ascending or descending order, for example, as long as both sender and receiver assume the same order. Second, the data structure must support an efficient search mechanism. In performing the computation, it is not enough for a processor to receive nonlocal data; it must also be able to fetch the correct nonlocal data to satisfy an array reference. The most convenient way to achieve this is to allow searches of the $recv_set(p,q)$ data structure and to store pointers into the appropriate message buffers in that structure.

5.2 Representing the Communication and Iteration Sets

Subsections 5.1.1 and 5.1.2 placed a number of constraints on data structures for the communication and iteration sets. The iteration set data structure must support two operations:

1. *Insertion* of unique elements (in the inspector)
2. *Traversal* of all stored elements (in the executor)

Communication set data structures have more demanding requirements:

1. *Insertion* of possibly nonunique elements (in the inspector)
2. *Transposition* to compute *send_set* from *recv_set* (in the inspector)
3. *Traversal* of all stored elements (in the executor)
4. *Search* of the stored elements (in the executor)

Of these constraints, those derived from the executor are more critical, since the inspector will only be performed once while the executor may be executed repeatedly. In addition, both data structures should be space-efficient given the limited storage available on each processor. Because of the difference in requirements, we use different data structures for the two types of sets. The subsections below describe our choices and examine some alternative designs.

5.2.1 Iteration Sets

A simple list suffices to represent the iteration sets. Because ordering is not important, the list can be built by insertions at the end of the list and read in the order it was created. Implementing the list as a dynamically-allocated array uses $O(N)$ creation and traversal time and $O(N)$ space for an N-element set. All of these bounds are theoretically optimal for representing an arbitrary set, and the constant factors in the implementation are quite small.

We can improve the storage efficiency of the list by observing that for many algorithms the nonlocal iterations tend to cluster into contiguous ranges. To take advantage of this, we keep a list of ranges rather than a list of individual elements. Since each range can be represented by its upper and lower bounds, the new data structure requires twice as much memory per list element, but the number of elements is presumably much smaller. Similarly, the time bounds on performance are affected by small constant factors.

In addition to our empirical observations on a relatively small set of applications, a probabilistic case can be made that ranges do well in the average case. The basic observation is that if $local_iter(p)$

```
record
   low: integer;              -- lower bound of range
   high: integer;             -- upper bound of range
   from_proc: integer;        -- sending processor
   to_proc: integer;          -- receiving processor
   array_id: integer;         -- array identifier
   buffer: ^real;             -- pointer to message buffer
end;
```

Figure 5.3: Communication set record

is sparse, then $nonlocal_iter(p)$ is dense, and vice versa. Sparse sets generate short ranges, which increase storage requirements; dense sets do the opposite. If each iteration randomly falls into $local_iter(p)$ with probability p_{local}, then elementary statistics gives the average length of a range of $local_iter(p)$ as $1/(1-p_{local})$ (assuming an infinitely long stream of iterations). Symmetrically, the average length of a range of $nonlocal_iter(p)$ is $1/(1-(1-p_{local})) = 1/p_{local}$. Thus, two adjacent ranges, one from each set, will contain $1/(1-p_{local}) + 1/p_{local}$ elements on average. If $0 < p_{local} < 1$ then $1/(1-p_{local}) + 1/p_{local} \geq 4$, so the list of ranges will have two elements while the list of individual iterations would have at least four. Thus, on average the space cost of the range list is no more than a list of individual elements.

5.2.2 Communication Sets

The operations of insertion, search, and traversal have been widely studied in the context of sequential machines. Three broad classes of data structures have been developed for these algorithms:

1. Lists (including both linked lists or dynamic arrays)
2. Search trees
3. Hash tables

These data structures and algorithms are applicable to the communication sets, because each processor applies the operations to only its local data. Because it is a problem unique to parallel machines, the transposition operation has been less studied. We have chosen to work with dynamically-allocated sorted arrays. The bulk of this section describes this data structure in some detail. A short comparison with search trees and hash tables appears at the end of the section.

Each element of the array representing a communication set is a record consisting of six fields, as shown in Figure 5.3. Like the iteration sets, array elements are grouped into contiguous ranges of subscripts; each range is represented by its upper and lower bound in the first two record fields. In order to collapse multi-dimensional array subscripts into a single subscript, the offset of the array element within its local section is used rather than the global subscript; this also avoids pathological behavior when processors' $local(p)$ sets are not contiguous. The next two fields name the processors which will send and receive the message. On processor p, all elements in the *send_set* data structure will have their *from_proc* fields set to p, and the *to_proc* fields in the *recv_set* data structure will also be p. This apparently wasted space is used in the transposition algorithm and cannot easily be removed. A field identifying the array allows the same data structure to be used if multiple arrays are accessed by the **forall**. Finally, a pointer field marks the base address of the range in the message buffer. The *recv_set* array is sorted on the *from_proc* field, with the *array_id* and *low* fields being secondary and tertiary fields, respectively. This allows $recv_set(p,q)$ for all q to be stored in a single array on processor p. Similarly, the *send_set* array is sorted by the *to_proc* field.

Subscripts can be inserted into the *recv_set* data structure by performing a binary search of the array. If the subscript is already part of a range, nothing further need be done. Otherwise, a new

range is added to the array unless the new subscript can be merged into an existing range. This is possible when the subscript being inserted is one less than an existing lower bound or one more than an existing upper bound. In either case, the appropriate bound is adjusted rather than adding a new range. This may also necessitate merging two adjacent ranges by adjusting the bounds of one range and deleting the other. If N is the number of array references and R is the number of ranges stored, the cost of one binary search is $O(\log R)$ and the cost of inserting a range is $O(R)$. The other computations can be done in constant time. Since N searches and $O(R)$ range insertions are done, the time for creating the *recv_set* data structure is $O(N \log R + R^2)$. In order to retrieve an array element using this data structure, a binary search of the array is performed to find the correct range, after which the buffer pointer can be used as the base offset of an array. Thus, the time to access a single array element is $O(\log R)$. The array elements can, of course, be accessed in sorted order by looping through the array at a cost of $O(N)$.

The transposition of *recv_set* into *send_set* can be performed on machines with a hypercube topology by using the Crystal routing strategy [FJL+86]. Assume that there are $P = 2^d$ processors numbered starting from 0, and that each processor p can communicate with processor q if p differs from q by exactly one bit. Call the processor which differs in the bit k the kth neighbor. The algorithm works in d stages. Initially, each processor stores its *recv_set* in the *send_set* array. At stage k, each processor removes from this array all entries for which the *from_proc* field differs from its own id in the kth bit. These entries are sent to the kth neighbor, and the corresponding message from that neighbor is received and added to the *send_set* array. Thus, after stage k, bits 0 through k match the processor id; after d stages, all of the bits match. Thus, processor p has records for all the messages it must send out. In other words, it has formed $send_set(p, q)$ as required. This is the algorithm used by the Kali run-time environment.

On machines without hypercube connectivity, similar algorithms can be developed for performing the transposition. On a two-dimensional mesh with $P = m*m$ processors, for example, $O(m)$ rounds of messages with a processor's north, south, east, and west neighbors would suffice. On machines supporting messages between arbitrary pairs of processors, the following algorithm can be used. Let the processors be numbered from 1 to P. Each processor p declares an array Msg_p of P integers, and sets $Msg_p[q]$ to 1 if $recv_set(p, q) \neq \phi$ and 0 otherwise. A tree summation is then performed to find $Msg[q] = \sum_{p=1}^{P} Msg_p[q]$ in $O(\log P)$ steps. Note that $Msg[q]$ gives the number of messages that processor q must send. All processors receive a copy of the Msg array. Each processor p then sends a message describing $recv_set(p, q)$ to each processor q for which the set is nonempty. It then receives $Msg[p]$ messages from other processors, which are precisely the descriptions of the messages it must send, i.e. $send_set(p, q)$ for various values of q. Processors from which no message is received have an empty *send_set*. This algorithm is general, but may not be efficient on some machines. In particular, mapping a binary tree onto the actual machine topology may require some nontrivial message routing, and sending the communication set messages may cause bottlenecks.

As stated above, our sorted arrays require $O(N)$ space and $O(N \log R + R^2)$ creation time, and provide $O(\log R)$ search and $O(N)$ traversal times for a list of N elements. In contrast, search trees typically require $O(N)$ space and have $O(N \log N)$ creation time, $O(\log N)$ search time and $O(N)$ traversal time. Because of the indirections used in implementing search trees, it appears that transposition is more complex than for dynamic arrays, but still possible. Hash tables require a fixed (but relatively large) amount of storage have $O(N)$ creation and $O(1)$ search times. It is not normally possible to traverse the elements of a hash table, although this can be added by maintaining an auxiliary list, which could then be used in our transposition algorithm. Our design decision to use dynamic arrays was based on several factors, including ease of implementation. The lack of a transposition algorithm for search trees was the major factor in deciding against search trees. Hash tables were discarded because of their high memory requirements; at the time we started the implementation, available memory was a severe limiting factor on the target machines. Some recent work by Mirchandaney et al. [MSMB90] has implemented a scheme similar to ours using hash tables; no groups have experimented with search trees.

```
processors Procs : array[ 1..P ] with P in 1..max_procs;
var A, Old_A : array[ 1..N ] of real dist by [ block ] on Procs;
    Count : array[ 1..N ] of integer dist by [ block ] on Procs;
    Adj : array[ 1..N, 1..max_nbrs ] of integer dist by [ block, * ] on Procs;
    Coef : array[ 1..N, 1..max_nbrs ] of real dist by [ block, * ] on Procs;

-- code to set up arrays 'Adj' and 'Coef' omitted

while ( not converged ) do

  -- code to copy A to Old_A omitted

  -- perform relaxation (computational core)
  forall i in 1..N on A[i].loc do
      var x : real;
      x := 0.0;
      for j in 1..Count[i] do
         x := x + Coef[i,j] * Old_A[ Adj[i,j] ];
      end;
      if (Count[i] > 0) then A[i] := x; end;
  end;

  -- code to check convergence omitted

end;
```

Figure 5.4: Nearest-neighbor relaxation on an unstructured grid

5.3 A Practical Example: Unstructured Mesh Relaxation

In this section we apply our analysis to the program in Figure 5.4. This models a simple partial differential equation solver on a user-defined mesh. Arrays A and *Old_A* store values at nodes in the mesh, while array *Adj* holds the adjacency list for the mesh and *Coef* stores algorithm-specific coefficients. This arrangement allows the solution of partial differential equations on irregular meshes, and is common in practice. We will only consider the computational core of the program, since it is the only part shown which requires communication.

The reference to $Old_A[Adj[i,j]]$ in this program creates a communication pattern dependent on data ($Adj[i,j]$) which cannot be fully analyzed by the compiler. Thus, the $ref(p)$ sets and the communication sets derived from them must be computed at run-time. Therefore, we use the inspector-executor strategy. Figures 5.5 and 5.6 show high-level descriptions of the code generated for the inspector and the executor, respectively. Sections 2.2.4 and 2.2.5 describe how the implementation iterates over $exec(p)$ and tests $i \in local_{Old_A}(p)$. The sets are stored as lists, as explained in Sections 5.2.1 and 5.2.2. Here, *local_list* stores $local_iter(p)$; *nonlocal_list* stores $nonlocal_iter(p)$; and *recv_list* and *send_list* store all of the $recv_set(p,q)$ and $send_set(p,q)$ sets for processor p, respectively. In Figure 5.6, access_list stores the search structure for accessing elements in $recv_set(p,q)$. We show it separately to emphasize the difference between the description of the messages and their contents. The statements in the **if** statement in Figure 5.5 compute the communication and iteration sets by examining every reference made by the **forall** on processor p. As discussed in Subsection 5.1.1, this conditional is only executed once and the results saved for future executions of the **forall**. The statements in Figure 5.6 are direct implementations of the code in Figure 5.2, specialized to this example. The locality test in the nonlocal computations loop is necessary because even within the same iteration of the **forall**, the reference $Old_A[Adj[i,j]]$ may be sometimes local and sometimes nonlocal. We will discuss the performance of this program in Chapter 6.

```
-- Code executed on processor p
if ( first_time ) then        -- only analyze once
   -- Initialize lists
   local_list := φ;  nonlocal_list := φ;
   send_list := φ;  recv_list := φ;
   -- Transformation of original forall
   for i ∈ exec(p) do
         flag := true;
         -- Generate recv_set(p,q)
         for j ∈ {1,2,...,Count[i]} do
            -- Check Old_A[Adj[i,j]]
            if ( Adj[i,j] ∉ local_A(p) ) then
                   Add Adj[i,j] to recv_list
                   flag := false;
            end;
         end;
         -- Generate iteration sets
         if ( flag ) then
             Add i to local_list
         else
             Add i to nonlocal_list
         end
   end;
   -- Generate send_set(p,q) from recv_set(p,q)
    Global communication phase to transpose recv_list into send_list
   first_time := false;
end;
```

Figure 5.5: Inspector for Figure 5.4

```
-- Code executed on processor p

 Send messages on send_list to their processors

for i ∈ local_list do
   var x : real;
   x := 0.0;
   for j ∈ {1,2,...,Count[i]} do
         x := x + Coef[i,j] * Old_A[ Adj[i,j] ];
   end;
   if (Count[i] > 0) then A[i] := x; end;
end;

 Receive messages on recv_list and store in access_list

for i ∈ nonlocal_list do
   var x : real;
   x := 0.0;
   for j ∈ {1,2,...,Count[i]} do
         -- Access nonlocal array element
         if ( Adj[i,j] ∈ local_A(p) ) then
            tmp := Old_A[ Adj[i,j] ];
         else
            Search access_list for Old_A[Adj[i,j]] and store in tmp
         end;
         x := x + Coef[i,j] * tmp;
   end;
   if (Count[i] > 0) then A[i] := x; end;
end;
```

Figure 5.6: Executor for Figure 5.4

5.4 Other Optimizations

Two important optimizations can be made in the inspector section of run-time analyzed **foralls**. Both involve caching of information to avoid recomputation. The executor is less amenable to optimization, but some improvement is also possible there.

The first inspector optimization involves groups of arrays within a single **forall** which all have the same reference pattern, which means that all arrays have identical distributions and references to them have the same subscripts. This is a typical situation for partial differential equation solvers, which use the same template for several variables. In this case the communication and iteration sets for the arrays will be identical, so it is only necessary to inspect one array's references. The sets can then be copied and used directly for each array, or the copying can be avoided by adding an extra level of indirection in the data structure.

A similar optimization can be used when two **foralls** have the same data access pattern. For this to occur, the index sets and all nonlocal references in the **foralls** must be the same. This is sometimes the case for multi-phase algorithms. When coupled with the first optimization, this also applies to solving systems of partial differential equations when each equation is solved by the same technique. The optimization in this case is simply to reuse the generated sets.

The major overhead in the executor is the search invoked to satisfy nonlocal references. Since run-time analysis is most likely to be used when the subscripts are irregular, it is unlikely that their calculation can be optimized significantly by the compiler. Common subexpression elimination, however, can reduce the number of searches significantly if the same subscript is used repeatedly. Note that in order to perform this optimization, the compiler must be able to prove that the search has no side effects. Providing this knowledge is not a problem if the run-time environment is integrated in the compiler, but is more difficult if the analysis is produced by a preprocessor. Depending on the exact nature of the data structures, it may be possible to reuse the results of a single search for other arrays with the same distribution pattern. This optimization is akin to combining locality checks for separate arrays in the inspector.

An analogue of another inspector optimization, saving the communication and iteration sets between **forall** executions, is also possible. In this case the information being saved is the results of the searches into the nonlocal access list. For every reference made dynamically in the **forall**, a pointer to the correct memory address (found by searching the communication set data structure) is saved on this list. This list can be constructed once in the inspector. Details of the inspector data structures, particularly when and how memory is allocated, determine whether this can be done as the communication sets are being constructed or if it requires a separate pass. Once the list is constructed, the executor can implement nonlocal references by an indirection from this list, followed by an advance to the next list element. Note that this arrangement requires the list to be generated in the same order as the references will be used. This requirement may interfere with other optimizations discussed above, particularly conservative approximations of control flow and combining the analysis for several arrays. Such a list is also expensive in terms of space. For these reasons, we did not implement this optimization in the Kali compiler. Some researchers have implemented such enumeration lists by hand [KMSB90, SCMB90, Lit90] and report excellent performance.

Chapter 6

Experimental Results

In order to evaluate the effectiveness of the methods described in Chapters 4 and 5, we incorporated them into the Kali compiler. This chapter describes several experiments validating and using that compiler. Section 6.1 describes the compiler and the experiments themselves, including the techniques used to time the compiled programs. Section 6.2 and 6.3 then describe the experimental results on programs using compile-time and run-time analysis, respectively.

6.1 Methodology

We first give an overview of the Kali compiler and its limitations. The prototype Kali compiler is written in C [KR88], with parsing being handled by Lex and Yacc [Joh75, LS75]. It consists of over 26000 lines of source code, not including approximately 19000 lines defining the abstract data types used to represent the parsed Kali programs. Of this, approximately 5200 lines are devoted to implementing the **forall** construct, including generating the communication statements. The compiler consists of two passes, one for parsing and one for generating code. The target language for the compiler is C extended with message-passing routines for the specific target machine. At present, code can be generated for the NCUBE/7 [HMS+86] or the iPSC/2 [PL88] computers. For brevity, however, we will only report results from the iPSC/2. We expect that extensions to any machine with a message-passing library would be quite straightforward.

Because the current compiler is a research prototype, it has certain limitations. The most significant limitation the compiler places on Kali programs is that processor arrays may only have one dimension. This implies that **forall** statements only need a single index, and generally simplifies their implementation. Data arrays may have any number of dimensions, but because of this limitation only one dimension can be distributed by the built-in distribution patterns. (User-defined distributions can, of course, depend on as many dimensions as the user desires.) Another important feature that is not implemented is nonlocal write accesses. As discussed in Section 3.6, this allows us to avoid a communication phase at the end of the **forall**. Both of these limitations will be corrected in later versions of the compiler.

The compiler automatically determines whether to use compile-time or run-time analysis on a **forall** by examining the subscripts accessed in the loop body. Currently only the compile-time analysis for Theorems 4.1, 4.4, and 4.8 is implemented (i.e. constant subscripts and subscripts of the form $f(i) = i + c$ for **block** and **cyclic** distribution patterns). Subscripts in undistributed dimensions are constrained to produce at most a one-dimensional array slice in the communication sets. This implies that at most one dimension in the constant subscript case can have a non-constant subscript, and no dimension in the other cases can have such a subscript. If all references in a **forall** meet these criteria, then compile-time analysis is used; if not, run-time analysis is generated for all references, including those that otherwise could have been analyzed in the compiler. Compile-time analysis generates code to evaluate the expressions defined in the appropriate theorems in Chapter 4.

These expressions must be evaluated at run-time because the number of processors is determined at load time rather than in the compiler. Run-time analysis is implemented as described in Chapter 5, except for the advanced techniques of Section 5.4.

Once the compiler was written, we tested both the compile- and run-time analysis routines for several attributes:

1. Absolute cost to generate the sets.

2. Absolute overhead of nonlocal array accesses.

3. Relative overheads within the context of a particular algorithm.

To do this, several programs were written in Kali and compiled into C code. Absolute costs were measured by copying nonlocal array elements, while the relative overheads were obtained from computational kernel algorithms. Details of these programs will be presented in Sections 6.2 and 6.3; here, we will only describe the methods used to time the programs. All of the times reported later are the averages of at least five runs. The methodology for each run is described next.

Before timing was started, all processors were forced to do a barrier synchronization to reduce variance due to waiting. Timings were then obtained by calls to the system clock before and after the appropriate computation. This limited the precision of our measurements to the granularity of the system clock (approximately 1 millisecond). To measure overall time, the calls were inserted directly in the Kali program. To obtain the times for sections of the code, such as time for communications only, we hand-modified the generated C code to remove all unnecessary sections. Because of the granularity of the timers, it was not possible to reposition the system calls to count only the appropriate sections of code; the computation in many sections was simply too short to measure. Our chosen methodology, however, has its own disadvantages. In particular, the iPSC/2 has data and instruction caches; deleting code changes cache behavior, generally by improving the cache hit ratio. This tends to reduce the times reported for partial computations, making the time for the entire computation greater than the sum of the partial computations' times. Counteracting this tendency is the possible overlap of computation and communication.

6.2 Experiments with Compile-time Analysis

6.2.1 Absolute Overheads

To measure the absolute overhead of compile-time analysis, we timed the copying program shown in Figure 6.1. This program tests the formulas of Theorem 4.4, which were the most computation-intensive ones we implemented. The function *iargv* is used to read command-line arguments to the program; this allows us to set $SIZE$, $OFFSET$, and IT to new values without recompiling the program. Setting $OFFSET = 0$ generates no nonlocal references, and other values for $OFFSET$ generate from 1 to $SIZE$ nonlocal references. In all cases, only processor 1 is accessing data, and this was the only processor for which we obtained timings. The copying is repeated IT times to avoid clock granularity problems. For an outline of the code generated for Figure 6.1, see Figure 4.6 in Section 4.3. The calculations for Figure 6.1 are slightly more complex than shown there because constant folding is not possible, but the principles are identical.

Three timings were obtained from Figure 6.1, each for various values of the parameters. First, we measured the absolute cost of generating the communication and iteration sets by timing the sections of the program which calculated the set descriptions. These sections correspond to the calculations in the **const** section and the allocation of the temporary array in Figure 4.6. The resulting measurements showed that the constant calculations required under 100 microseconds on the iPSC/2. For comparison, the message latency on that machine is 350 microseconds, and the communications bandwidth is 2.8 Mbyte/sec [Arl88]. Thus, the cost of calculating communication and iteration sets is insignificant in comparison to the cost of actually performing the communication

```
processors procs : array[ 1..NP ] with NP in 1..128;

const
   SIZE : integer = iargv(1);          -- number of array elements per processor
   OFFSET : integer = iargv(2);        -- size of shift in access loop
   IT : integer = iargv(3);            -- number of iterations
   N = SIZE * NP;                      -- total size of array

var
   x, y, index : array[ 1..N ] of integer dist by [ block ] on procs;

for ii in 1..IT do

   -- access nonlocal elements from processor 1
   forall i in 1..SIZE on x[i].loc do
         x[i] := y[ i + OFFSET ];
   end;

end;
```

Figure 6.1: Kali program for basic compile-time measurements

(unless, of course, the sets are empty and no message is sent). This is quite encouraging, since very little effort was put into optimizing the performance of these calculations.

To measure the overhead of accessing nonlocal array elements, we timed the copying loops with no communications. This corresponds to timing the **for** loops in Figure 4.6 marked "local computations" and "nonlocal computations." To measure a baseline, we set $OFFSET$ to zero, giving no nonlocal references. We then ran several tests with $OFFSET$ set to values greater than $SIZE$, which ensured that all references would be nonlocal. To get a fair picture of the overheads involved, we forced the compiler to generate a locality test for references in the nonlocal iteration loop, as described in Section 4.5. The results of these tests showed that a single local reference cost approximately 3.33 microseconds while the nonlocal reference cost 6.23 microseconds. It is clear that the locality test is a significant overhead. This is mitigated somewhat by the fact that the majority of iterations will be local in a well-designed program. Future research on reducing this overhead is clearly needed, however.

6.2.2 Realistic Performance

As a realistic example of a program amenable to compile-time analysis, we chose Gaussian elimination without pivoting. Figure 6.2 shows the exact program used. Figure 6.3 shows a Kali translation of the generated C code. The form of the broadcast is generated from the formulas of Theorem 4.1, while the bounds on the temporary array are taken from the **for** j loop. Cyclic array distribution was used to achieve good load balancing. The copying loop and outermost **for** loop are not essential features of the algorithm; they were added to simplify checksums and to avoid running afoul of the system clock, respectively. They are included here for completeness.

The Gaussian elimination program was run for several values of N and using all the possible machine sizes. For each combination, three timings were obtained: the *total time* for the program, the time for *computation* only, and the time for *communication* only. Copying the pivot row into the temporary array was included in the communication time. Note that this will result in an apparent communication overhead even for a single processor. The constant IT (the number of times the elimination was performed) was always chosen to keep the times well above the clock granularity, and the raw times were divided by IT to normalize them to one execution of the algorithm. Table 6.1 gives the results. All times in the table are in seconds. The communication

```
processors
   proc : array[ 1..NP ] with NP in 1..128;

const
   N : integer = iargv(1);          -- size of matrix (from command line)
   IT : integer = iargv(2);         -- number of iterations

var
   a : array[ 1..N, 1..N ] of double dist by [ cyclic, * ] on proc;

for ii in 1..IT do

   -- copy ax to a
   forall i in 1..N on a[i,1].loc do
      for j in 1..N do
            a[i,j] := ax[i,j];
      end;
   end;

   -- gaussian elimination without pivoting
   for k in 1..N-1 do
         forall i in k+1 .. N on a[i,1].loc do
            for j in k+1 .. N do
                  a[i,j] := a[i,j] - a[k,j] * a[i,k] / a[k,k];
            end;
         end;
   end;

end;
```

Figure 6.2: Kali program for Gaussian elimination

```
-- Code on processor p

var
   a : array[ 1..N, 1..N ] of double dist by [ cyclic, * ] on proc;
   temp : array[ 1..N ] of double;            -- compiler temporary

for ii in 1..IT do

   -- copy loop omitted

   for k in 1..N-1 do
         -- communications statements: broadcasting
         if ( k%NP = p ) then
            temp[ k..N ] := a[ k, k..N ];
            send( temp1[k..N], procs[*] );
         else
            temp[ k..N ] := recv( procs[*] );
         end;
         -- computation statements
         for i in k+1 + (p-k-1)%NP .. N by NP do
            for j in k+1 .. N do
                  a[i,j] := a[i,j] - temp[j] * a[i,k] / temp[k];
            end;
         end;
   end;

end;
```

Figure 6.3: Compiled form of Figure 6.2

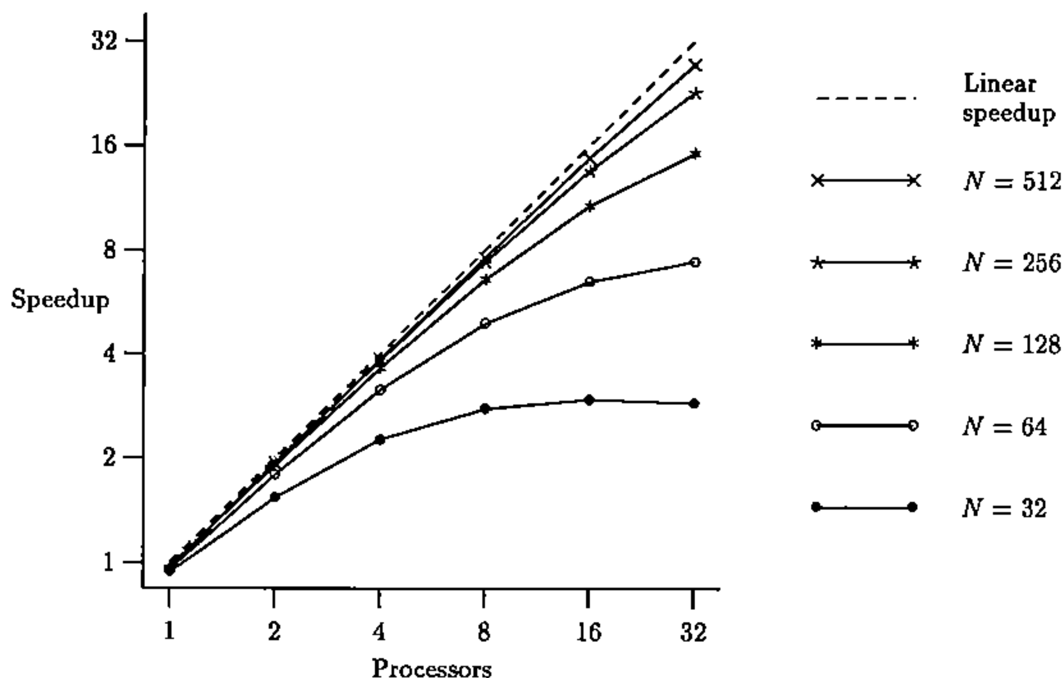


Figure 6.4: Speedup of Gaussian elimination program

time for one processor reflects the time to compute the communication sets and copy the pivot row for sending (although no message is actually sent). The copying is overhead which does not appear in the original program, but it may ultimately save time in the computation section; it allows the pivot row to be accessed as a one-dimensional array rather than as a two-dimensional array. Table 6.1 also gives the speedup, defined as the total computation time divided by the computation time on one node. Given the available data, this compares the Kali program to the "best" sequential program. Memory restrictions did not allow the 512×512 matrix factorization to be run on one processor. In that case, the single-processor time for calculating speedup was estimated from its operation count and the single-processor computation time for the 256×256 problem. The speedup figures are also presented graphically in Figures 6.4.

The Kali program does not achieve perfect speedup for any entry in Tables 6.1 for two reasons:

1. The computation time does not scale linearly. In this case, there is no overhead for locality checking in the generated programs; the deviation from linear speedup is entirely due to imperfect load balancing. This effect becomes negligible for larger problem sizes.

2. The communication overhead is significant, particularly when the number of rows per processor is small. In a sense, this is inherent in the algorithm; any implementation which uses distributed data will need to communicate between processors.

Any parallel program would have the communication overhead, but might avoid load balancing problems. We therefore calculated "perfect" parallel times by adding the measured Kali communication time to the single-processor computation time divided by the number of processors. These times served as a realistic comparison to the actual Kali run times. The results of this comparison are shown graphically in Figure 6.5. Note that the Kali programs are very close to the "perfect" times in all of the graphs, in many cases being indistinguishable. This is more clearly shown in Figures 6.6, which graph the ratio of the Kali program times to the perfect program times. In those figures, the shape of the curves is less important than the vertical scale; it indicates that the Kali programs

Performance for $N = 32$				
Processors	Total time	Computation	Communication	Speedup
1	0.2959	0.2773	0.0095	0.93
2	0.1809	0.1458	0.0347	1.53
4	0.1229	0.0760	0.0466	2.25
8	0.0999	0.0412	0.0589	2.77
16	0.0941	0.0233	0.0708	2.94
32	0.0963	0.0143	0.0826	2.88
Performance for $N = 64$				
Processors	Total time	Computation	Communication	Speedup
1	2.339	2.231	0.030	0.95
2	1.252	1.157	0.088	1.78
4	0.710	0.591	0.118	3.14
8	0.456	0.307	0.148	4.88
16	0.344	0.165	0.178	6.47
32	0.302	0.096	0.208	7.39
Performance for $N = 128$				
Processors	Total time	Computation	Communication	Speedup
1	18.894	18.205	0.107	0.96
2	9.568	9.274	0.238	1.90
4	5.007	4.670	0.313	3.63
8	2.769	2.373	0.386	6.57
16	1.696	1.231	0.459	10.72
32	1.199	0.729	0.534	15.17
Performance for $N = 256$				
Processors	Total time	Computation	Communication	Speedup
1	152.39	147.58	0.40	0.96
2	76.55	75.37	0.72	1.92
4	38.65	37.59	0.91	3.81
8	19.94	18.78	1.10	7.39
16	10.89	9.52	1.29	13.54
32	6.45	5.71	1.49	22.85
Performance for $N = 512$				
Processors	Total time	Computation	Communication	Speedup
2	611.91	606.71	2.38	1.93
4	307.24	303.17	2.95	3.85
8	155.70	151.59	3.53	7.60
16	80.34	75.91	4.09	14.73
32	43.10	38.32	4.67	27.47

Table 6.1: Performance of Gaussian elimination program

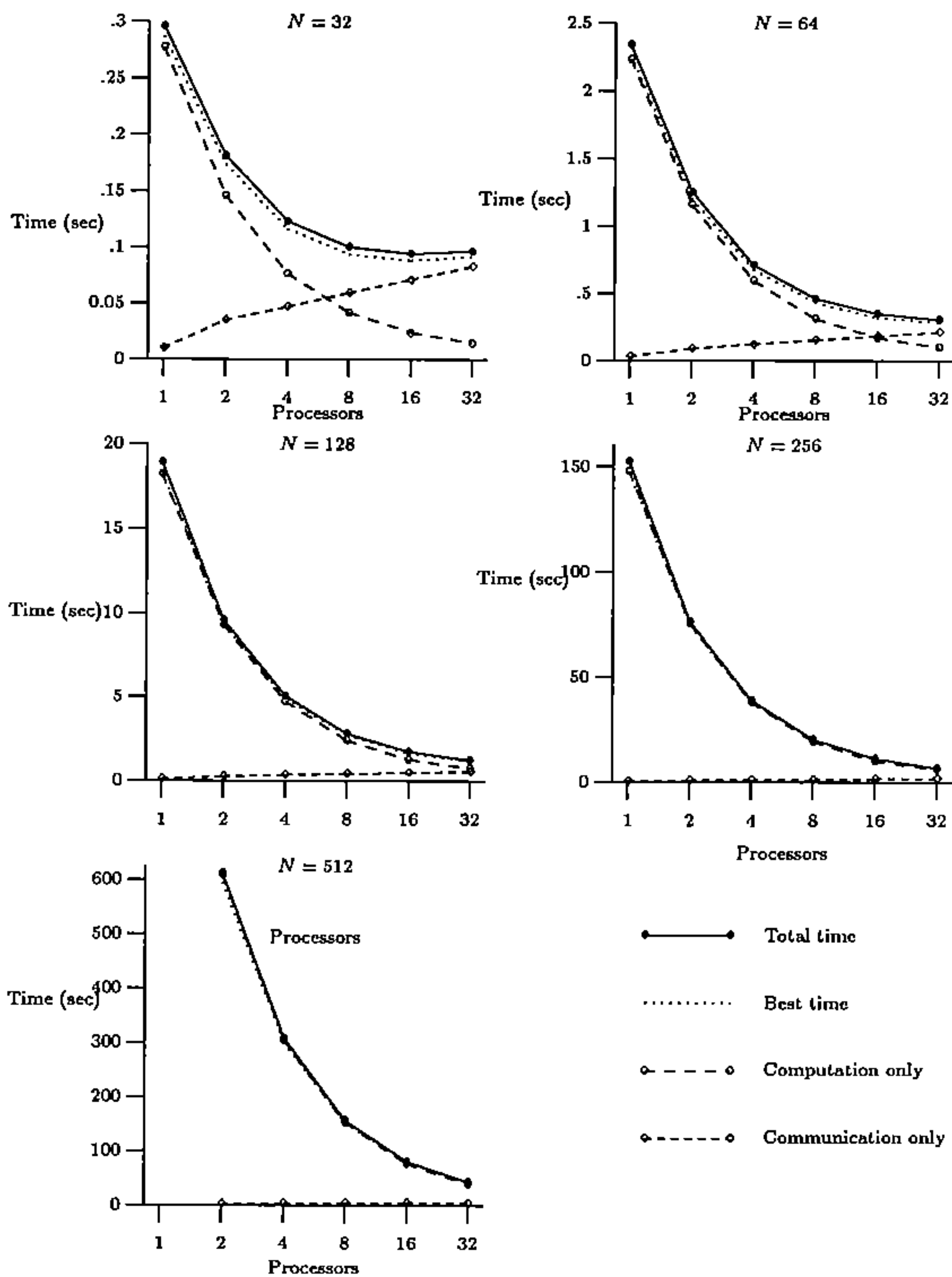


Figure 6.5: Comparison of Kali and "perfect" parallel programs for Gaussian elimination

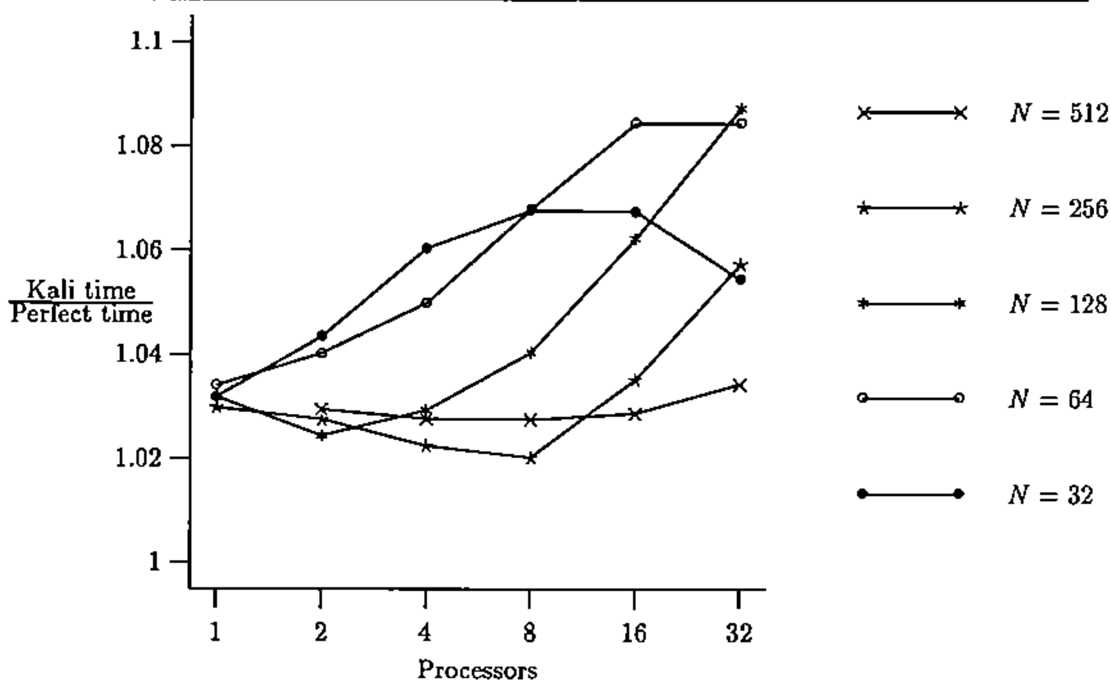


Figure 6.6: Overheads of Kali programs for Gaussian elimination

never have an overhead of more than 9% on the iPSC/2. This indicates that Kali is performing nearly as well as the best possible implementation on this example.

6.2.3 Conclusions

The results of our experiments with compile-time analysis can be summarized as follows:

1. The time to generate the communication and iteration sets using compile-time analysis is quite small, particularly when compared to the time to pass a message.
2. Programs with many nonlocal references may pay a significant overhead for locality checks if the implementation described in Section 4.5 is used. This is in addition to the overhead of sending and receiving the messages, which will occur regardless of the implementation used.
3. Compile-time analysis produces practical code for at least one major algorithm, Gaussian elimination. The code is nearly optimal relative to a simple model of program performance.

The conclusion to be drawn from these observations is that compile-time analysis is a promising approach to allowing high-level programming of nonshared memory machines. Where it can be applied it generates the necessary communications quickly and, in many cases, provides efficient access to nonlocal data.

6.3 Experiments with Run-time Analysis

6.3.1 Absolute Overheads

To measure the absolute overhead of run-time analysis, we timed the copying program shown in Figure 6.7. The index array can be set to create any behavior from all references local to all references nonlocal. Additionally, the distribution of the nonlocal references can be changed; this

```
processors procs : array[ 1..NP ] with NP in 1..128;

const
   IT = 5;                               -- number of iterations
   RANGES : integer = iargv(1);          -- number of ranges per processor
   SIZE : integer = iargv(2);            -- number of array elements per processor
   N = SIZE * NP;                        -- total size of array

var
   x, y, index : array[ 1..N ] of integer dist by [ block ] on procs;

for ii in 1..IT do

   -- access nonlocal elements from processor 1
   forall i in 1..SIZE on x[i].loc do
         x[i] := y[ index[i] ];
   end;

end;
```

Figure 6.7: Kali program for basic run-time measurements

is useful because our data structure for the communication sets is sensitive to this distribution. Figure 6.8 outlines the code generated for Figure 6.7.

Our first experiments with run-time analysis studied the cost of the inspector. We first generated five programs from the code of Figure 6.8:

1. *Basic-insp*, which included only lines 1 through 5 and line 13. This represented the lowest-level overhead of the inspector.

2. *Check*, which included only lines 1 through 6, 8 and line 13.

3. *Insert*, which included only lines 1 through 8 and line 13.

4. *Iterate*, which included all lines except line 12.

The modified programs allowed us to compute the times for all major operations in the inspector.

1. The time for the *locality checks* only is the difference between *Check* and *Basic-insp*.

2. The time for *insertions into the receive list* is the difference between *Insert* and *Check*.

3. The time for *insertions into iteration sets* is the difference between *Iterate* and *Insert*.

4. The time for *global communications* is the difference between the entire inspector and *Iteration*.

We timed these programs and the full inspector, using several sets of values for index, and computed the above measures. In all cases, every reference was nonlocal; the settings of index varied the number of ranges representing the communication sets from 1 to 200. $SIZE$ was varied from 5000 to 30000 to allow expressions for the performance based on time to be computed.

The performance of the inspector can be summarized as follows:

1. The time for one locality check is 8.17 microseconds on the iPSC/2.

2. The time to insert one element in the the receive list depends logarithmically on the number of ranges in the set. More precisely, the time is $9.36\log_2(R) + 30.8$ microseconds.

```
-- Code executed on processor p

if ( first_time ) then                                              -- Line 1

   local_list := φ;  nonlocal_list := φ;                            -- Line 2
   send_list := φ;  recv_list := φ;                                 -- Line 3

   for i ∈ local_x(p) ∩ {1, ..., SIZE} do                           -- Line 4
         flag := true;                                              -- Line 5
         if ( index[i] ∉ local_y(p) ) then                          -- Line 6
            Add index[i] to recv_list                               -- Line 7
            flag := false;                                          -- Line 8
         end;
         if ( flag ) then                                           -- Line 9
            Add i to local_list                                     -- Line 10
         else
            Add i to nonlocal_list                                  -- Line 11
         end
   end;

   Global communication phase to transpose recv_list into send_list -- Line 12
   first_time := false;                                             -- Line 13

end;

 Send messages on send_list to their processors

-- local iterations
for i ∈ local_list do
   x[i] := y[ index[i] ];
end;

 Receive messages on recv_list and store in access_list

-- nonlocal iterations
for i ∈ nonlocal_list do
   if ( index[i] ∈ local_y(p) ) then
         tmp := y[ index[i] ];
   else
         Search access_list for y[index[i]] and store in tmp
   end;
   x[i] := tmp;
end;
```

Figure 6.8: Compiled form of Figure 6.7

3. The time to insert one element in the iteration lists is 5.54 microseconds.

4. The time for the global communication phase depends (linearly) on the number of ranges and (logarithmically) on the number of processors. For this program the dominating factor is the number of ranges. Based on this observation, a good expression for predicting performance is $27.0R + 2170$ microseconds.

All of these overheads combined give the total inspector time. A predicting expression for this time is

$$T = 8.17A + 5.54I + (9.36\log_2(R) + 30.8)N + 27.0R + 2170$$

where

$$\begin{aligned} A &= \text{Number of array references in } \textbf{forall} \\ I &= \text{Number of iterations of } \textbf{forall} \\ N &= \text{Number of nonlocal array references in } \textbf{forall} \\ R &= \text{Number of ranges in the receive list on one processor} \end{aligned}$$

and the resulting time is given in microseconds. This expression was obtained by combining the analyses above.

It is difficult to compare directly the costs of computing the lists in the inspector with the communication required by the inspector, because the performance of two aspects depend on different factors. The total computation time in the inspector is dominated by the time to insert elements in the receive list and iteration lists, which depends on the numbers of ranges and iterations as well as the number of nonlocal references. The communication cost depends on the number of processors and the number of ranges. In this example it was computation that was the deciding factor. This would not be the case if the number of ranges were nearer the number of references, however; in that case, even the model given would be invalid, since the quadratic insertion time in the range list would come into play. In any case, however, it appears that the inspector would be of the same order of magnitude as the cost of the actual **forall** computation on a given number of processors if the **forall** were doing significant processing on each iteration, rather than simply copying array elements. This assessment is borne out in the next section on a more realistic example.

Our next experiment with run-time analysis concerned the overhead of accessing nonlocal data. We ran two series of tests, one which accessed only local data and one in which all references were nonlocal, and timed the for loops in the executor. To eliminate the loop overhead, we also ran another test with empty loop bodies and subtracted the time found there. The number of ranges representing the communication sets in the nonlocal tests ranged from 1 to 200. From the local tests, we obtained times for one local access of 3.9 microseconds on the iPSC/2. (This differs from the results in the last section because the costs of computing the subscripts themselves differ.) The nonlocal tests on both machines showed nonlocal access times that were logarithmic in the number of ranges, which was expected because we were using binary search. A more exact expression is

$$T = 9.02\log_2(R) + 23.8$$

where R is the number of ranges and T is the time to access a single nonlocal element on a given machine, given in microseconds. Thus, for example, accessing one element taken from 8 ranges would cost 41 microseconds. This is a high overhead, but not so high that it could not be overshadowed by computation time in the iteration. A single square root operation on the iPSC/2, for example, takes approximately 48 microseconds.

6.3.2 Realistic Performance

As a realistic example of a program requiring run-time analysis, we used the unstructured mesh relaxation program shown in Figure 6.9. Except for details of the outermost loop, this program is

```
processors
   proc : array[ 1..NP ] with NP in 1..128;
const
   N : integer = iargv(1);          -- mesh size (from command line)
   IT : integer = iargv(2);         -- number of iterations
   ANS : double = dargv(3);         -- check value
   NBRS = 6;                        -- max number of neighbors
var
   A, Old_A : array[ 1..N ] of real dist by [ block ] on proc;
   Count : array[ 1..N ] of integer dist by [ block ] on proc;
   Adj : array[ 1..N, 1..NBRS ] of integer dist by [ block, * ] on proc;
   Coef : array[ 1..N, 1..NBRS ] of real dist by [ block, * ] on proc;

for ii in 1..IT do
   -- copy A to Old_A
   forall i in 1..N on A[i].loc do
         Old_A[i] := A[i];
   end;
   -- nearest neighbor relaxation on a
   forall i in 1..N on A[i].loc
         var x : double;
   do
         x := 0.0;
         for j in 1..count[i] do
            x := x + Coef[i,j] * Old_A[ Adj[i,j] ];
         end;
         if (Count[i] > 0) then A[i] := x; end;
   end;
end;
```

Figure 6.9: Kali program for unstructured mesh relaxation

identical to Figure 5.4 in Section 5.3. Figures 5.5 and 5.6, also in that section, outline the inspector and executor generated for this program.

We tested the program on several grids. Here we will focus on one typical example, a random modification of a square mesh with 4 nearest-neighbor connections. The base mesh is referred to as a "5-point star" in the literature; the modified mesh is designed to model unstructured meshes. To modify the mesh, 10% of the edges were randomly reset to other points. The points of the original mesh were numbered to map a horizontal strip of points onto each processor (i.e. the one-dimensional distribution simulates a two-dimensional distribution blocked by rows). We will refer to this mesh as the "modified square" mesh. The meshes were created with torus wrap-around connections, thus giving each point exactly four neighbors. We varied the number of points in the mesh from 2^{10} to 2^{18} (corresponding to meshes of dimension 32×32 to 512×512), and adjusted IT (the number of iterations) for each mesh to avoid clock granularity issues.

For each mesh size, we obtained five timings.

1. The *total time* to execute the program.

2. The time for the *inspector* only.

3. The time for the *executor* only.

4. The time for the *computation* in the executor.

5. The time for the *communication* in the executor.

The raw data for the iPSC/2 are given in Table 6.2. The nonzero communication times for one node are attributable to checking the (empty) message lists and to clock granularity. Because the inspector is only executed once, the different values of IT would make a straightforward calculation of the parallel speedup deceiving. Therefore, we have normalized the times used in computing speedup to assume $IT = 100$ for all values of N. For experiments which used smaller values of IT, this was done by adding the inspector time to 100 times the time for one executor sweep. The value used for sequential time was the computation time on one processor, if available; if a given mesh size would not fit on one node, the sequential time was extrapolated from the sequential time for the largest mesh which did fit. Figure 6.10 graphs the number of processors against the parallel speedup.

As with the Gaussian elimination program, times for the unstructured mesh solver do not achieve perfect linear speedup. This can be attributed to three sources of overhead:

1. The time to execute the *inspector*

2. The *communication* time in the executor

3. The search overhead in *accessing nonlocal references*

The communication overhead is inherent in the algorithm, while the inspector and nonlocal access overheads are artifacts of our implementation. To take the inherent overhead into account in evaluating our program, we proceed as in Section 6.2.2 by comparing actual Kali performance to a "perfect" parallel program consisting of linear speedups in the computation added to our actual communication time. The results of this comparison are shown graphically in Figure 6.11. The times were again normalized to 100 mesh sweeps to allow comparison between different sized meshes. Unlike the Gaussian elimination experiments, the difference between the Kali times and the "perfect" times is noticeable at some points in all the graphs. Figure 6.12 shows this overhead can be nearly 100% on the iPSC/2. Figures 6.13 and 6.14 break down this overhead into its inspector and nonlocal access components. Note that the apparent exponential increase in the last two graphs is caused by the logarithmic scale on the horizontal axis; in reality, the increases are closer to linear relations.

Figure 6.13 illustrates the inspector overhead by plotting the ratio of the inspector time and the "perfect" parallel time versus number of processors. This can be interpreted as the number of extra **forall** sweeps that the inspector is costing. For example, a ratio of 2 means the inspector is

Performance for $N = 1024$, $IT = 100$						
Processors	Total time	Inspector time	Executor			Speedup
			Total	Comp.	Comm.	
1	1.644	0.012	1.631	1.630	0.002	0.992
2	0.939	0.008	0.930	0.851	0.072	1.737
4	0.599	0.007	0.593	0.449	0.139	2.718
8	0.395	0.007	0.388	0.245	0.139	4.120
16	0.291	0.007	0.284	0.148	0.140	5.590
32	0.239	0.007	0.233	0.094	0.140	6.792

Performance for $N = 4096$, $IT = 100$						
Processors	Total time	Inspector time	Executor			Speedup
			Total	Comp.	Comm.	
1	19.683	0.110	19.572	19.570	0.001	0.994
2	10.703	0.069	10.629	10.382	0.170	1.829
4	5.984	0.044	5.946	5.565	0.296	3.267
8	3.544	0.032	3.512	3.146	0.324	5.522
16	2.360	0.027	2.707	1.973	0.328	7.160
32	1.748	0.026	1.724	1.370	0.330	11.183

Performance for $N = 16384$, $IT = 25$						
Processors	Total time	Inspector time	Executor			Speedup
			Total	Comp.	Comm.	
1	24.090	0.525	23.565	23.564	0.000	0.994
2	12.741	0.296	12.439	15.435	0.058	1.883
4	7.622	0.168	8.313	7.811	0.093	2.820
8	4.541	0.102	4.670	3.510	0.093	5.018
16	2.721	0.072	2.680	2.496	0.097	8.733
32	1.685	0.057	1.703	1.419	0.098	13.719

Performance for $N = 65536$, $IT = 10$						
Processors	Total time	Inspector time	Executor			Speedup
			Total	Comp.	Comm.	
2	18.680	0.990	17.810	17.147	0.035	1.745
4	9.881	0.537	8.762	8.891	0.049	3.545
8	5.461	0.309	4.814	4.705	0.049	6.451
16	2.756	0.195	2.932	2.561	0.051	10.587
32	1.869	0.138	1.819	1.452	0.053	17.053

Performance for $N = 262144$, $IT = 10$						
Processors	Total time	Inspector time	Executor			Speedup
			Total	Comp.	Comm.	
8	24.644	1.275	23.412	22.154	0.078	6.818
16	13.550	0.747	13.409	11.765	0.077	11.903
32	8.083	0.466	7.685	6.872	0.078	20.757

Table 6.2: Performance of unstructured mesh relaxation program

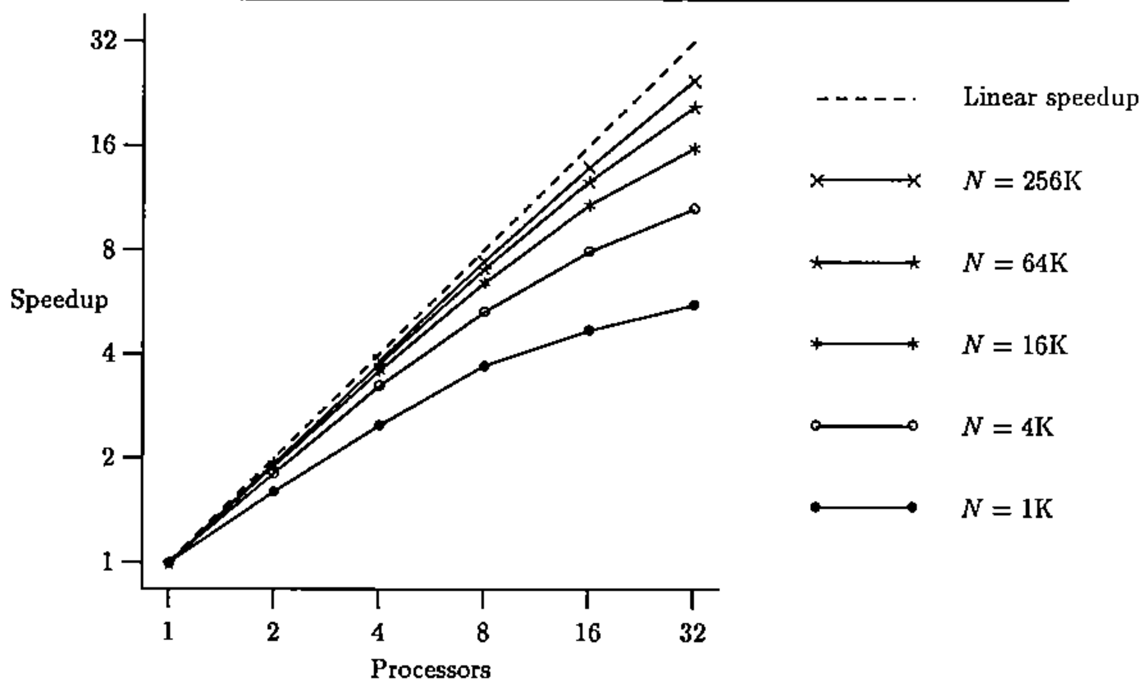


Figure 6.10: Speedup of unstructured mesh relaxation program

effectively adding two passes through the **forall** to the total time; this is a 2% overhead if the **forall** is repeated 100 times, or a 200% overhead if the **forall** is executed once. The figure shows that for this example the inspector is never more expensive than 2.4 perfect parallel sweeps on the iPSC; this is probably acceptable if the **forall** is executed many times. It is also important to note that the relative inspector overhead decreases as the problem size increases. This is important because a major reason for adding processors to a parallel system is to solve larger problems; in such cases, the inspector becomes even more attractive.

Figure 6.14 shows the nonlocal access overhead by plotting the ratio of executor computation time to the computation time assuming linear speedup. Since the program data is constructed to be perfectly load balanced, any difference in timings is due to the search invoked on nonlocal accesses. The overhead relative to the above perfect parallel times will be less than the ratios shown in these graphs because of the effect of communications. Even so, the computation overheads here are large; up to 211%. It should be noted, however, that the nonlocal access overhead is also inversely related to problem size. For the largest problem, the overhead is only 10% on 32 processors. This indicates that our search technique may be acceptable for large problems, which require the most computation time in any case.

The results of experiments using other classes of meshes were similar to those above. As an indication of this, Figure 6.15 shows the speedup curves for four representative meshes. The "9-Point Star" mesh is a regular mesh connecting each point to its eight nearest Cartesian neighbors. Points were numbered to simulate the two-dimensional block distribution of Section 3.2.3; therefore, each processor must communicate with at most eight others. The "Square" mesh connects each point to its four Cartesian neighbors. The "Hexagonal" mesh tiles the plane with hexagons rather than squares and connects adjacent hexagons; the "Modified Hexagonal" mesh varies this by randomly deleting 10% of the edges. The points in the last three meshes were numbered to map horizontal strips of the mesh onto the same processor; thus, communications for the regular meshes were confined to the two neighboring processors. Of these meshes, the 9-point star and modified hexagonal meshes probably provide the most realistic models of actual unstructured meshes, because they produce larger numbers of ranges in the communication sets. The fact that each processor communicates

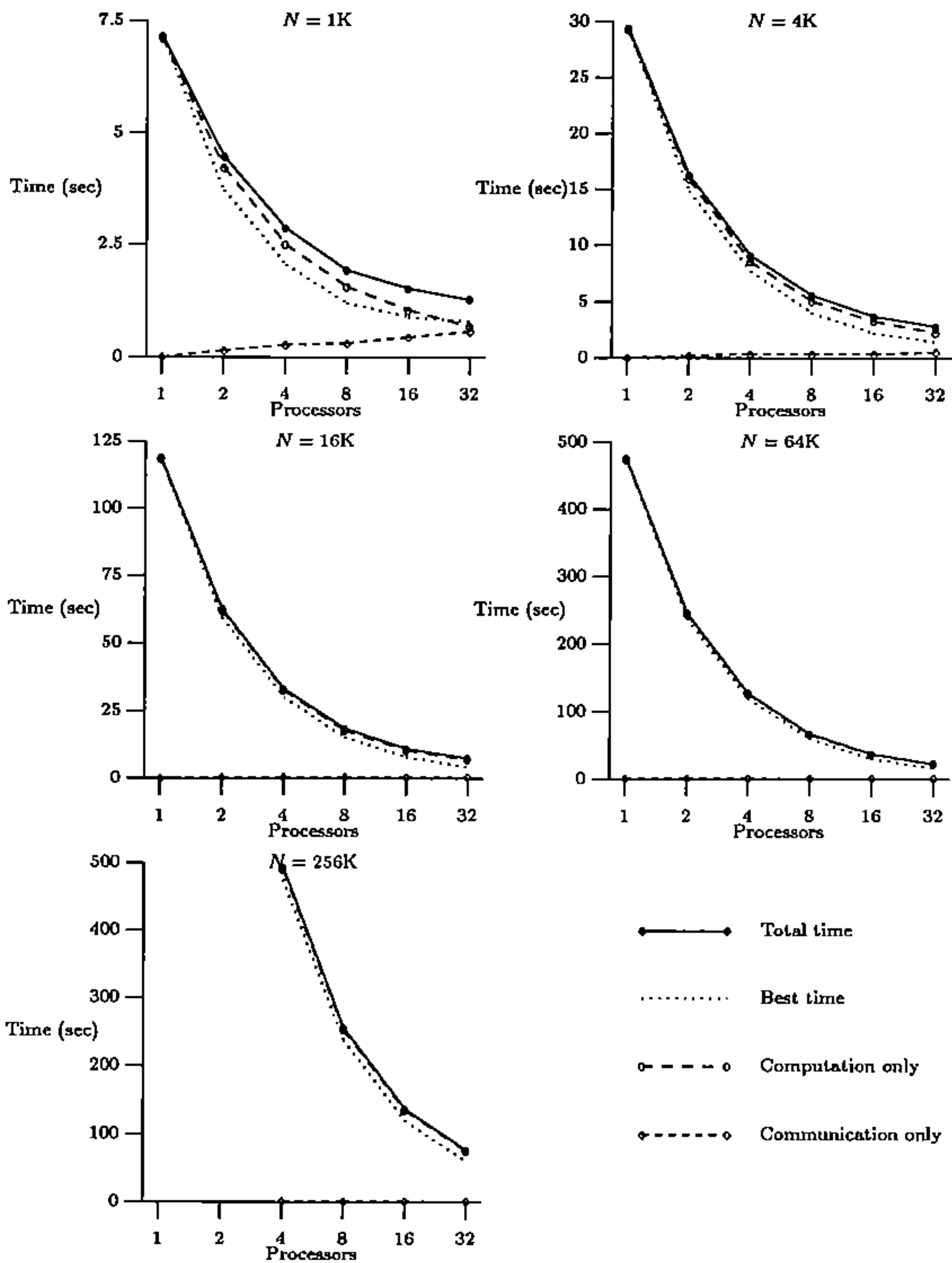


Figure 6.11: Comparison of Kali and "perfect" parallel programs for unstructured mesh relaxation

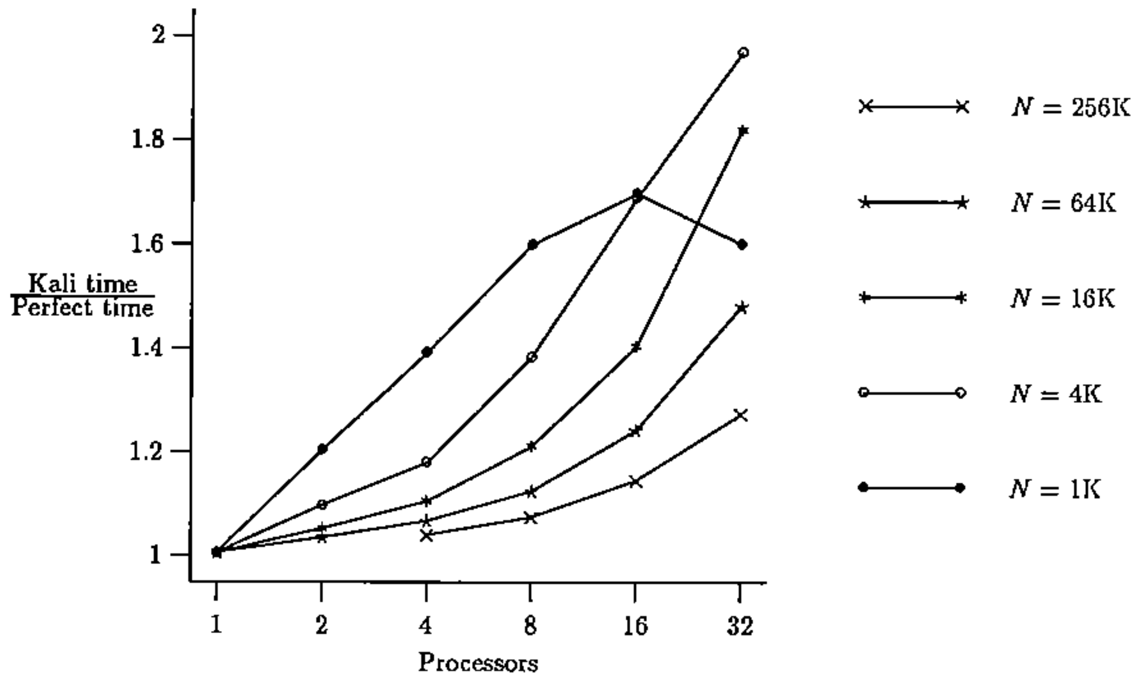


Figure 6.12: Overheads of Kali programs for unstructured mesh relaxation

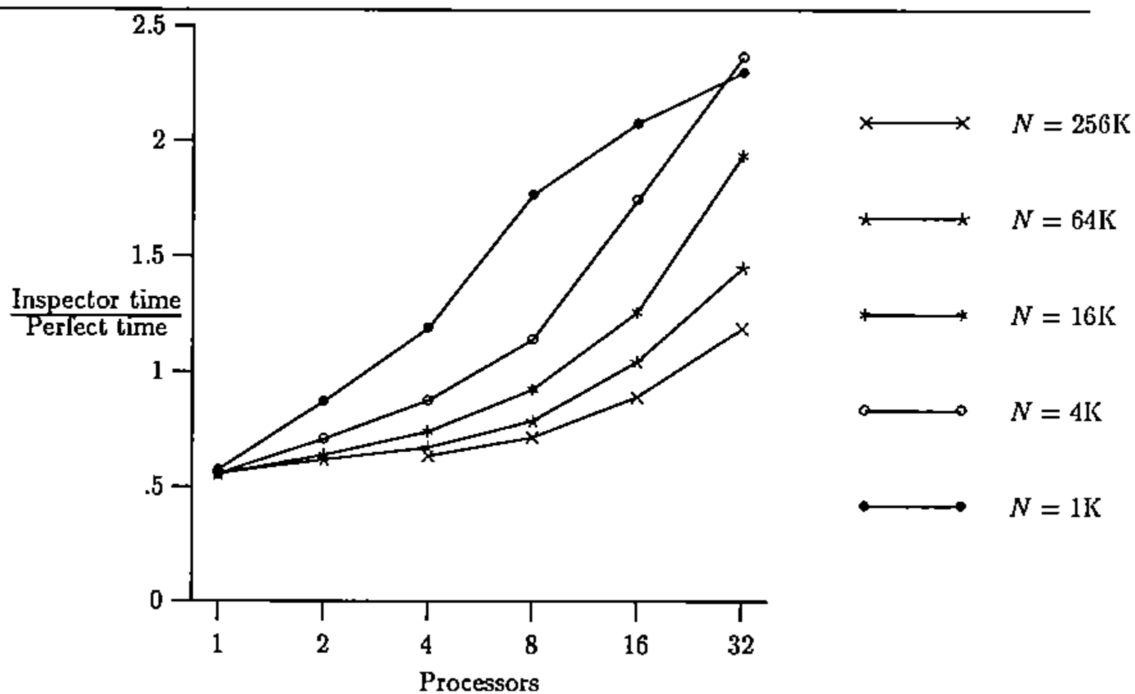


Figure 6.13: Inspector overheads in unstructured mesh relaxation

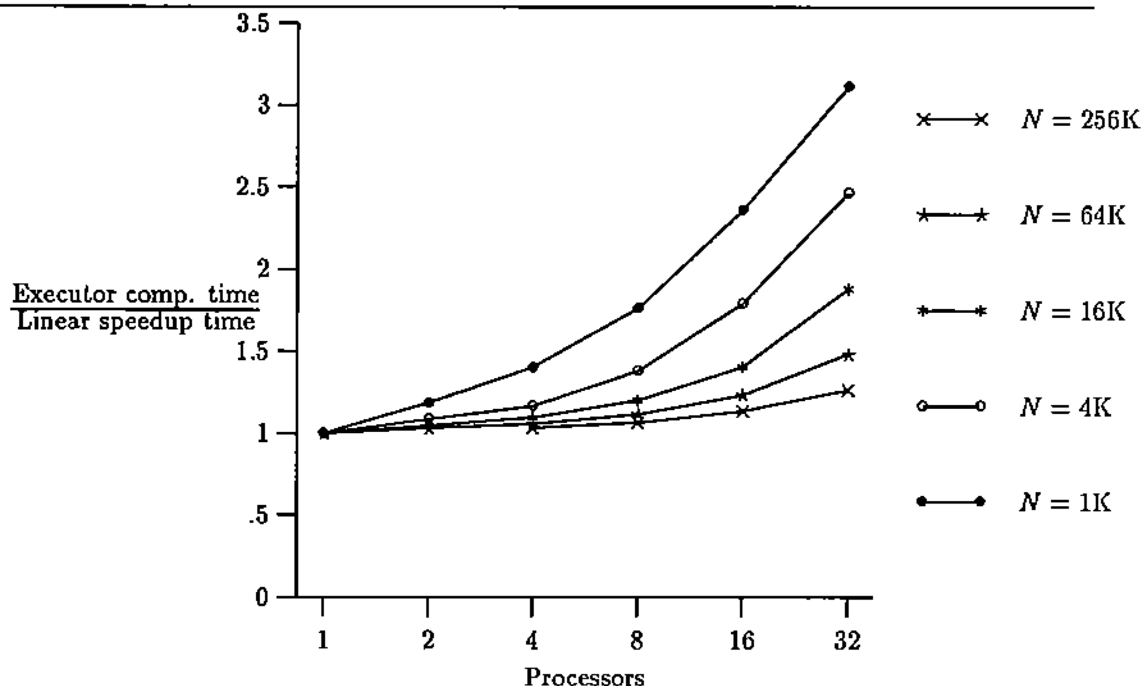


Figure 6.14: Nonlocal access overheads in unstructured mesh relaxation

with a limited number of others is realistic, since sensible data distribution will produce this situation. The curves are very similar to Figure 6.10 in general outline.

6.3.3 Conclusions

The results of our experiments with run-time analysis can be summarized as follows:

1. The time required for the inspector is equivalent to a few extra executions of the **forall** in the optimal parallel program. Whether this is acceptable depends on the number of times the **forall** is executed and on the availability of alternative implementations.
2. The overhead of performing nonlocal references is very high when there are many of them. This is usually the case when the amount of data on a single processor is small, either because the problem size is small or because many processors are used on a fixed-size problem.
3. Of the two overheads, the more important is the nonlocal reference overhead. It is relatively larger and cannot be amortized as the inspector overhead can be.
4. Both the inspector and nonlocal access overheads scale very well with problem size. Although less apparent from the experiments, it can be expected that the overheads will also decrease as the amount of computation in the **forall** increases. This is because the overheads are roughly proportional to the number of nonlocal accesses, while additional computation is unlikely to require proportionally more data.

The conclusion to be drawn from these observations is that run-time analysis is a promising approach for large-grain problems which cannot be analyzed by the compiler. Smaller problems incur relatively large overheads. The size of problems for which run-time analysis is effective can be expected to decrease for more computation-intensive problems. This allows high-level programming of nonshared memory machines for a large class of problems, although there are still problems for which the analysis is not effective.

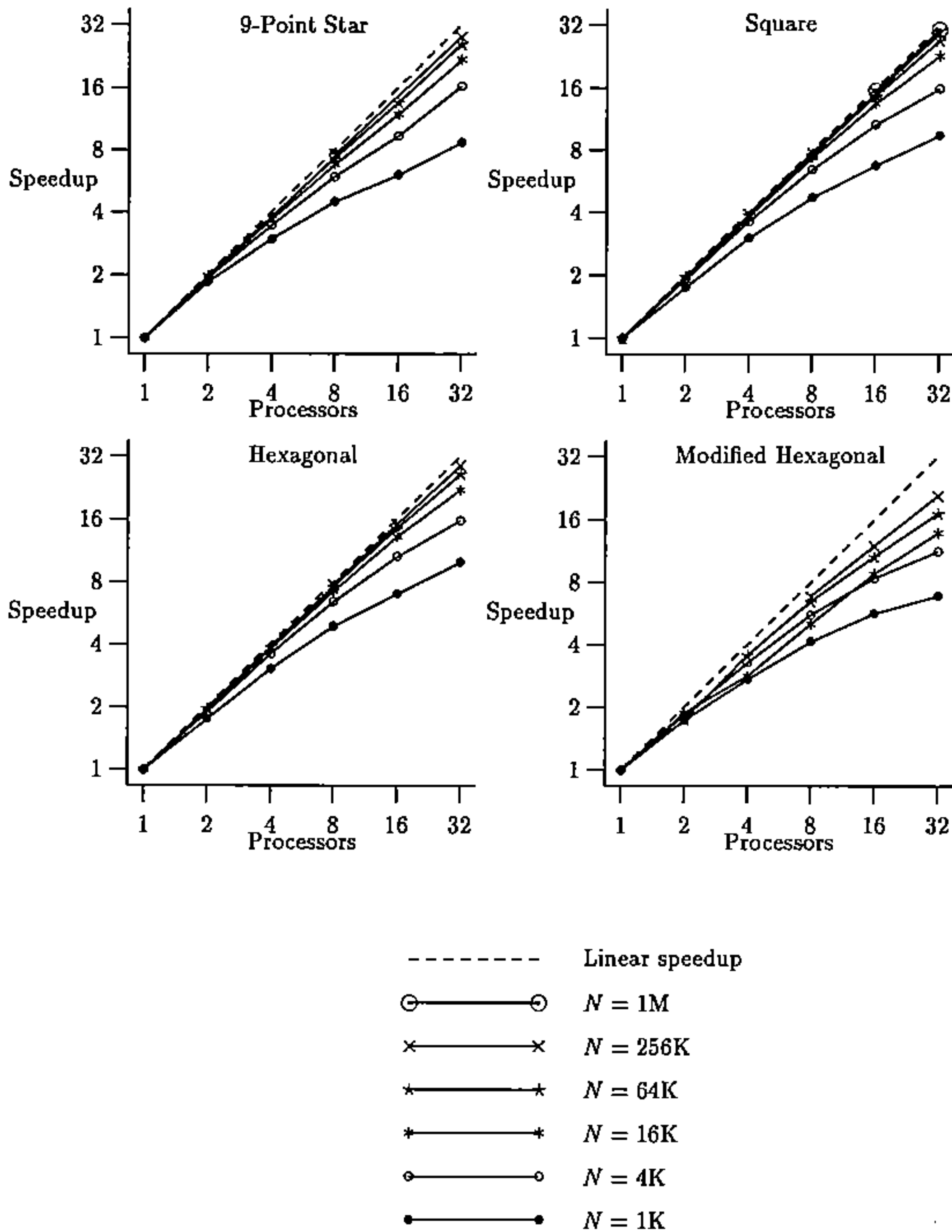


Figure 6.15: Speedup curves for other unstructured meshes

Chapter 7

Conclusions

7.1 Summary of Contributions

This work has made significant contributions in four general areas:

1. It has developed a *formal model* for mapping shared-memory programs onto nonshared-memory machines.
2. It has derived a *compile-time analysis* applicable to a large class of programs from the formal model.
3. It has developed a *run-time strategy and environment* which is generally applicable from the formal model.
4. It has validated the compile-time and run-time results by *incorporating them in a compiler*.

The model of Chapter 3 is a useful general framework for modeling both the compilation and the execution of shared-memory programs on nonshared-memory machines. Ours is the first model to unify the research done on the problems of compile- and run-time mapping techniques for programs on these machines. In the past, most research in parallelizing transformations has not used an explicit model, making it difficult to evaluate and compare different groups' work. Explicit models like ours will help to change this situation, as well as driving research in developing new techniques. They may also eventually be useful in evaluating individual programs to provide feedback to the user; at present, no standard means of evaluation is available.

The compile-time analysis of Chapter 4 provides the best evidence to date of the utility of our model. The formulas derived there allow efficient programming of nonshared-memory machines using shared-memory models when they can be applied. The code generated even by a first implementation is very close to optimal according to a simple model of program performance.

The run-time analysis of Chapter 5 provides a promising alternative approach to mapping shared-memory programs to nonshared-memory machines for cases where compile-time analysis cannot be used. The overheads of run-time analysis are higher than compile-time analysis, but run-time techniques have no limits on their applicability. Our experiments indicate that these overheads are certainly acceptable for large problems, and may be acceptable for smaller problems if no other alternative is available. Furthermore, we have presented evidence that run-time analysis will improve in efficiency as more complex algorithms are used, making it doubly attractive for realistic work.

The implementation of compile- and run-time analysis in the Kali compiler demonstrates that programs based on shared-memory models can be automatically mapped onto nonshared-memory machines. Run-time analysis ensures the generality of the compilation process, while compile-time analysis achieves high efficiency in many common situations. The language thus allows the programmer to concentrate on the higher-level aspects of the program, often without sacrificing performance.

Such a combination of ease of programming and efficient execution will make nonshared memory machines usable by a much wider audience. Kali is one of the first programming languages able to claim this advantage.

7.2 Related Research

There is a great deal of current research which is related in some way to ours, including general parallel programming languages and language extensions [Ame83, Gel85, Jor86, McG82, MSA+85, MV87], transformations for exploiting parallelism [Wol82, ACK86, ABC+87], and shared virtual memory systems [BCZ90, CAL+89, Li86, RAK88]. In this section we will examine only the most closely related work.

Callahan and Kennedy [CK88] and Zima, Bast and Gerndt [ZBG88] suggest transforming annotated sequential FORTRAN into message-passing code by inserting explicit send and receive statements at the array-element level. Vectorization techniques are then aggressively applied to aggregate messages, avoiding the overheads for small messages. The annotations to their programs describe the data distribution in much the same way that Kali **dist** clauses do. In a sense, their work is a bottom-up approach in contrast to our top-down approach. It appears that for cases amenable to our compile-time analysis, their work produces essentially the same code. Our run-time analysis appears to be beyond the abilities of their transformations, so our techniques hold an advantage there. On the other hand, the bottom-up approach also applies to more general parallel loops (such as the *doacross* [Cyt84]) which we do not consider. The two approaches therefore have complementary strengths.

Gerndt [Ger89] extends the above work with the concept of "overlap" between the sections of arrays stored on different processors and shows how it can be automatically computed. As explained in Section 4.5, this allows programs compiled with his methods to avoid locality checks during execution. A further advantage of this method is to make nonlocal data persistent between **forall** statements, avoiding some communication overhead. In terms of our framework he has generalized our analysis to distribution functions with overlap (as shown in Section 3.2.3). Automatic computation of the overlaps corresponds to choosing the distribution function in the compiler rather than having it specified by the user. His results are not without their limitations, however; in the worst case, every processor will store every element of the array. Two cases in which this would occur are Gaussian elimination and unstructured mesh relaxation. The techniques presented here do not have such pathological space overheads.

Pingali and Rogers [Rog90, RP89], working with the functional language Id Nouveau, have developed a compilation scheme very similar to those of Callahan and Kennedy and Zima and his coworkers. They explicitly consider run-time resolution of messages, but do not retain information between **forall** executions as our run-time analysis does. They also extend the bottom-up approach by applying their version of compile-time analysis to forms of parallel loops which pipeline computation and communication. Another notable difference between our implementation and theirs is that they produce a separate program for each processor, while we only generate one program. Their strategy allows much more constant folding and perhaps other optimizations, but ours produces programs which are independent of the number of processors in the target machine.

Schnabel, Weaver, and Rosing [RSW89, RSW90] have developed DINO, the language most similar to Kali that we have found. Instead of Kali's processors, DINO has a construct called an environment. Several environments may be mapped to the same processor, but this incurs a performance penalty. DINO has user-specified data distributions like Kali, but allows data to be mapped to multiple environments and does not support arbitrary user-defined data mappings. Communication is implicit in DINO in the sense that message-passing statements are not needed, but nonlocal references within loops must be explicitly tagged by the programmer. Fully implicit communication is possible via subroutine parameters, in which entire array sections are automatically copied if necessary. This allows convenient specification of regular communication patterns, but the situation for irregular patterns is less clear. It is certainly possible to duplicate contiguous sections of arrays in

a given environment, but it is unclear whether disconnected regions can be duplicated and accessed efficiently. The current implementation of DINO apparently does not perform any equivalent to our compile- and run-time analysis on loops. This is undoubtedly because the need for such an operation is less severe in DINO, where communication can be aggregated by the user by array copy statements and procedure parameters. Future versions of DINO, however, promise to remove the tagging of nonlocal references; at that time such analysis will probably be added to the compiler.

André, Pazat, and Thomas [APT90] have implemented the Pandore language, which shares many goals with Kali. In Pandore, parallelism is automatically extracted from essentially sequential code, although a **forall** statement also exists. They have independently developed a bottom-up compilation approach similar to Kennedy and Zima. They stress regular problems; it is unclear how they handle irregular ones.

Tseng [Tse89] has developed the AL language and its compiler targeted to the WARP systolic array. His work stresses linear subscript functions for the distributed dimension of an array. (Only one dimension is distributed in AL because the WARP is a linear array.) His work is thus directly comparable to our compile-time analysis. An important advance over our work is that AL automatically generates the distribution for an array, including overlap information, given only the programmer's specification of which dimension is to be distributed. His distributions are also more general than our **block** and **cyclic** distributions. Detection of parallel loops is automatic in the AL compiler, and more general forms of loops are handled than our work considers. AL, however, does have its limitations. There is no equivalent to our run-time analysis for irregular problems, which would be extremely difficult to map onto a systolic array in any case. Because small messages are not as expensive on systolic architectures, the AL compiler does not need to aggregate messages as our compile- and run-time analysis does. Finally, the restriction of distributions to one dimension may not allow generalization of the distribution choice algorithm to multiple dimensions.

Chen and her colleagues [CCL89, LC89, LC90] have implemented the Crystal functional programming language on nonshared memory machines. In doing so they have developed a robust theory similar to our formalism in many ways, but based on lambda calculus models rather than sets. Their equivalent of distribution functions do not appear to be easily extensible to replicated data, however. A large portion of their work is concerned with automatically distributing data among processors, which is a significant addition to this line of research. The exact problem that they solve, however, is to determine sets of elements of different arrays which should be mapped to the same processor. It appears that this is orthogonal to choosing the distribution patterns themselves. This may partially account for the much lower speedups that they report for Gaussian elimination in [CCL89]. Another fundamental difference in approach is that they generate communication statements by matching patterns of subscripts with descriptions of synchronous message-passing routines. This has two disadvantages compared to our approach:

1. Computation and communication cannot be overlapped because of the synchronization in communications.

2. Patterns which do not match exactly result in extra data being communicated. This appears to be a particularly severe problem with the dynamic access patterns which our run-time analysis is designed to handle.

Compensating these to some extent is the fact that synchronous communication primitives can be more efficient than the asynchronous routines we use because they can avoid congestion.

Quinn and Hatcher [QH90] have implemented C* [RS86], a language originally designed for the Connection Machine, on the NCUBE/7. While some of their work is directly related to the SIMD semantics of the languages, the optimizations they apply to message-passing are closely related to our work. Quinn and Hatcher describe their optimizations as "vectorizing" the messages; this reflects the close relation of their work with the bottom-up approach described above. Like the other groups using this approach, they produce good code for regular communication patterns and bad code for other patterns.

Reeves [CR89] is also approaching programming nonshared-memory machines from the direction of SIMD languages. His Paragon system uses the same ideas as Quinn and Hatcher for regular problems, but he has also done work on irregular problems. In Paragon, a programmer can define a mapping function describing a data movement pattern. This function corresponds to f^{-1} in Section 3.4. The Paragon implementation has features allowing efficient application of this mapping. In effect, this arrangement gives the user an elegant interface for writing inspectors and executors. Reeves does not generate the mapping functions automatically, however.

Saltz and his coworkers [MSMB90, MSS+88, SBW90, SC86, SCMB90] have independently been pursuing run-time optimizations similar to those mentioned in Section 3.7. They use an inspector-executor strategy identical to ours, but use different data structures to represent the communication sets. Saltz reports on two different schemes:

1. A hash table scheme which is directly comparable to our sorted range lists.
2. A scheme which enumerates all references in the **forall** separately, which avoids even the locality test and hash table lookup overhead in the executor.

Both schemes have lower time overheads than ours, but require more memory overhead.) (The enumeration scheme is the extreme case for both these statements.) A clear advancement over our work is that Saltz considers the **doconsider** loop, which is essentially a **doacross** loop which must be scheduled dynamically. This is a more general construct than our **forall** statement, although it is handled by very similar methods. Early versions of this work used FORTRAN-callable subroutines to provide a convenient user interface to the inspector and executor; more recent work has produced a prototype compiler. Littlefield [Lit90] has independently studied a very similar scheme.

7.3 Directions for Future Research

The results reported here can be extended in many ways. We divide the extensions into six categories:

1. Extensions to the basic model of Chapter 3
2. Extensions to the compile-time analysis of Chapter 4
3. Extensions to the run-time analysis of Chapter 5
4. Implications for the programming interface to nonshared memory machines
5. New application areas for similar techniques
6. Heuristics for parallel compilation

The basic model of data distribution can already support distributions which replicate data, but the formulas of Sections 3.4 and 3.5 do not apply to those distributions. Adding these would allow more more efficient implementation of some algorithms. New formulas similar to Equations 3.20, 3.21, 3.27, and 3.28 are needed for parallel control structures besides the **forall**. Examples of such control structures include the **doacross** loop and functional decomposition techniques. The identification of index sets with the integers also should be relaxed to allow data structures other than arrays, such as trees. All of these appear to be relatively straightforward extensions.

Compile-time analysis can certainly be extended to other static distributions such as the block-cyclic and skewed distributions of Section 3.2. The forms of subscripts which can be analyzed should also be extended. Particularly interesting in this regard are multidimensional arrays with coupled subscripts (i.e. several dimensions which are functions of the same index). Because of the range of possible subscript forms and distributions, it is tempting to automate this process. This could be done by incorporating theorem-proving algorithms directly in the compiler, by providing an interface between the compiler and existing symbolic algebra systems, or simply by using symbolic algebra systems off-line to prove the necessary theorems. In any case, the result would be a larger class of

programs amenable to compile-time analysis. At some point even this aggressive program will fail, however, because of fundamental limits such as the undecidability of Diophantine equations. To avoid futile effort it would also be wise to prove theorems giving the limits of compile-time analysis. Both computability results and complexity results for various classes of subscripts would be useful in this regard.

The design and analysis of data structures for inspectors and executors will continue to be of interest for some time. Optimizations like those of Section 5.4 will also be vital for extracting performance from irregular codes. The optimizations may, of course, interact with the data structures used, further complicating the analysis of the data structures. Because of this complexity, providing user control over how the inspector and executor are implemented may be fruitful. Because of the complexity of these pieces of code, however, much work is needed on the interface.

All aspects of this research have implications for how a programmer should approach a nonshared memory machine. The basic model and machine-dependent information about the cost of compile- and run-time analysis should be used for performance prediction for programs. These predictions can be fed back to the programmer to enable efficient programming. This is particularly relevant to run-time analysis, where seemingly trivial changes to the source code can have a great impact on performance. Annotations should also be provided for the programmer to specify important information relevant to the analysis. The best example of this is complex subscripts. Often a programmer can prove that the referenced array element will be local, but the compiler cannot. In this case, an annotation can be used to prevent costly communication code from being generated. No groups mentioned in Section 7.2 have considered the implications of this research for debugging in any detail. It appears that programs are easier to write with shared-memory models, but the generated program is so far removed from the original that it is difficult to see how it can be debugged. This is a common problem with high-level languages and highly optimizing compilers, but no general solution is known. Finally, the mechanisms for run-time analysis suggest general implementation techniques for user data structures. Making the run-time data structures part of the descriptors for dynamically allocated arrays, for example, could provide an effect similar to overlapping distributions for those data structures; similar comments apply to nonnumerical data.

Applications of our techniques can be made to other classes of parallel machines and to other language models. There are several other classes of parallel computer in which data locality is important. Examples of such machines include the Connection Machine [TR88] and the BBN Butterfly [BBN87]. Models and compilation techniques similar to ours should yield performance improvements on these machines. First steps toward doing this were reported in [KMV87a, KMV87b]. Performance models like those mentioned above should also be applicable to these machines. As discussed above, the basic model should also be applicable to other parallel control constructs. In addition, it may be possible to apply the techniques to other programming paradigms such as functional programming and SIMD languages. Some of the groups mentioned in Section 7.2 are already pursuing these ideas independently.

One major limitation of the current Kali implementation is the presence of **dist** and **on** clauses. These clauses have no effect on the correctness of the code, but instead only affect performance. It would be better if the compiler chose the data distributions and location of computations automatically. This is a difficult problem, however. Mace [Mac83] shows that a form of the distribution choice problem is NP-complete, and location of computations can be modeled in the same way. Heuristics will therefore be needed for these operations. The last section describes some current work in this area.

Bibliography

[ABC^{+}87] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. Research Report RC 13115 (#56866), IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1987.

[ACK86] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. Computer Science Technical Report TR86-42, Rice University, Houston, TX, November 1986.

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[Ame83] American National Standards Institute, Inc. *Reference Manual for the Ada Programming Language,* ANSI/MIL-STD-1815A-1983 edition, February 1983.

[APT90] F. André, J.-L. Pazat, and H. Thomas. PANDORE: A system to manage data distribution. In *International Conference on Supercomputing,* pages 380–388, June 1990.

[Arl88] R. Arlauskas. iPSC/2 system: A second generation hypercube. In *Proceedings of the 3rd Conference on Hypercube Concurrent Computers and Applications,* pages 38–42, Pasadena, CA, January 19-20 1988.

[BBN87] BBN Advanced Computers, Inc., Cambridge, MA. *Butterfly Product Overview,* 1987.

[BCZ90] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pages 168–176, Seattle, WA, March 14-16 1990.

[CAL^{+}89] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, and R. J. Littlefield. The Amber system: Parallel programming on a network of processors. In *Proceedings of the 12th ACM Symposium on Operating Systems Priciples,* pages 147–158, December 1989.

[CCL89] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-760, Yale University, New Haven, CT, December 1989.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing,* 2:151–169, 1988.

[CR89] A. Chueng and A. P. Reeves. The Paragon multicomputer environment: A first implementation. Technical Report EE-CEG-89-9, Cornell University Computer Engineering Group, Ithaca, NY, July 1989.

[Cyt84] R. G. Cytron. *Compile-time Scheduling and Optimization for Asynchronous Machines.* PhD thesis, University of Illinois, Urbana, IL, October 1984.

[Fen81] T. Feng. A survey of interconnection networks. *Computer*, pages 12–27, December 1981.

[FJL+86] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Fly66] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54:1901–1909, December 1966. reprinted in R. H. Kuhn and D. A. Padua, *Tutorial on Parallel Processing*.

[FW78] S. Fortune and J. Willie. Parallelism in random access machines. In *Proceedings of the 10 ACM Symposium on Theory of Computing*, pages 114–118, San Diego, CA, 1978.

[GCKW79] D. I. Good, R. M. Cohen, and J. Keeton-Williams. Principles of proving concurrent programs in Gypsy. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, pages 42–52, San Antonio, TX, January 29-31 1979.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[Ger89] H. M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, December 1989.

[Gil58] E. N. Gilbert. Gray codes and paths on the n-cube. *Bell System Technical Journal*, 37:815, May 1958.

[GJG88] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Conference Proceedings of the International Conference on Supercomputing*, pages 238–253. ACM Press, July 1988.

[Han75] P. B. Hansen. The programming language concurrent pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, June 1975.

[HMS+86] J. Hayes, T. Mudge, Q. Stout, S. Colley, and J. Palmer. Architecture of a hypercube supercomputer. In Kai Hwang, Steven Jacobs, and Earl Swartzlander, editors, *Proceedings of the 1986 International Conference on Parallel Processing*, pages 653–660, August 1986.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[INM86] INMOS, Colorado Springs, CO. *A Toutorial Introduction to Occam Programming*, 1986.

[Joh75] S. Johnson. Yacc — yet another compiler compiler. CS Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[Jor86] H. Jordan. Structuring parallel algorithms in an MIMD, shared memory environment. *Parallel Computing*, 3(2):93–110, May 1986.

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

[KM89] C. Koelbel and P. Mehrotra. Compiler transformations for non-shared memory machines. In *Proceeding of the 4th International Conference on Supercomputing*, volume 1, pages 390–397, May 1989.

[KMSB90] C. Koelbel, P. Mehrotra, J. Saltz, and S. Berryman. Parallel loops on distributed machines. In *Proceedings of the 5th Distributed Memory Computing Conference*, page to appear, Charleston, SC, April 9-12 1990.

[KMV87a] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic domain decomposition in BLAZE. In Sartaj K. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 521–524. Pennsylvania State University Press, August 1987.

[KMV87b] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Semi-automatic process partitioning for parallel computation. *International Journal of Parallel Programminging*, 16(5):365–382, 1987.

[KMV90] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186, Seattle, WA, March 14-16 1990.

[Koe88] C. Koelbel. A formalism for describing data distribution. Technical Report CSD-TR 803, Purdue University, West Lafayette, IN, August 16 1988.

[Koe90] Charles Koelbel. *Compiling Programs for Nonshared Memory Machines.* PhD thesis, Purdue University, West Lafayette, IN, August 1990.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Eaglewood Cliffs, NJ, second edition, 1988.

[LC89] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Technical Report YALEU/DCS/TR-725, Yale University, New Haven, CT, November 1989.

[LC90] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. Technical Report YALEU/DCS/TR-755, Yale University, New Haven, CT, May 1990.

[Li86] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, New Haven, CT, September 1986.

[Lil90] S. L. Lillevik. Touchstone program overview. In *Proceedings of the 5th Distributed Memory Computing Conference*, page to appear, Charleston, SC, April 9-12 1990.

[Lit90] R. Littlefield. Efficient iteration in data-parallel programs with irregular and dynamically distributed data structures. Technical Report 90-02-06, University of Washington, Department of Computer Science and Engineering, Seattle, WA, February 1990.

[LS75] M. E. Lesk and E. Schmidt. *UNIX Programmer's Manual*, volume 2, chapter Lex — A Lexical Analyzer Generator. AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[Mac83] M. Mace. *Globally Optimal Selection of Memory Storage Patterns.* PhD thesis, Duke University, Durham, NC, May 1983.

[McG82] J. R. McGraw. The VAL language: Description and analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, January 1982.

[MMS79] J. G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual, version 5.0. Technical Report CSL-79-3, Xerox Palo Alto Research Center, April 1979.

[MSA+85] J. McGraw, S. Skedzielewski, S. Allan, R. Oldenhoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Language reference manual. Report M-146, Lawrence Livermore National Laboratory, March 1985.

[MSMB90] S. Mirchandaney, J. Saltz, P. Mehrotra, and H. Berryman. A scheme for supporting automatic data migration on multicomputers. ICASE Report 90-33, Institute for Computer Applications in Science and Engineering, Hampton, VA, May 1990. to appear in *Proceedings of the 5th Distributed Memory Computing Conference.*

[MSS+88] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, St. Malo, France, 1988.

[MV87] P. Mehrotra and J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

[MV89] P. Mehrotra and J. Van Rosendale. Compiling high level constructs to distributed memory architectures. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[PBG+85] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In D. Degroot, editor, *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771. Computer Society Press, August 1985.

[PL88] Douglas Pase and Allan Larrabee. *Programming Parallel Processors*, chapter Intel iPSC Concurrent Computer, pages 105–124. Addison-Wesley Publishing Company, 1988.

[QH90] M. J. Quinn and P. J. Hatcher. Compiling SIMD programs for MIMD architectures. In *Proceedings of the 1990 IEEE International Conference on Computer Language*, pages 291–296, March 1990.

[RAK88] U. Ramachandran, M. Ahamad, and M. Y. A. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-CS-88/23, Georgia Institute of Technology, Atlanta, GA, June 1988.

[RAP87] D. Reed, L. Adams, and M. Patrick. Stencils and problem partitioning: Their influence on performance of multiprocessor systems. *IEEE Transactions on Computers*, C-36(7):845–858, July 1987.

[Rog90] A. Rogers. *Compiling for Locality of Reference.* PhD thesis, Cornell University, Ithaca, NY, August 1990.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, June 21-23 1989.

[RS86] J. R. Rose and G. L. Steele. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, Cambridge, MA, 1986.

[RSW89] M. Rosing, R. W. Schnabel, and R. P. Weaver. Expressing complex parallel algorithms in DINO. In *Proceedings of the 4th Conference on Hypercubes, Concurrent Computers, and Applications*, pages 553–560, 1989.

[RSW90] M. Rosing, R. W. Schnabel, and R. P. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, University of Colorado, Boulder, CO, April 1990.

[SBW90] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and runtime compilation. ICASE report 90-59, Institute for Computer Applications in Science and Engineering, Hampton, VA, 1990.

[SC86] J. Saltz and M. Chen. Automated problem mapping: The crystal runtime system. In *Proceedings of the Hypercube Microprocessors Conference*, Knoxville, TN, 1986.

[SCMB90] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[Sei85] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–32, January 1985.

[SLY88] Z. Shen, Z. Li, and P. C. Yew. An empirical study on array subscripts and data dependencies. CSRD Report 840, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, December 1988.

[Sol90] K. Solchenbach. SUPRENUM: The European parallel supercomputer. In *Proceedings of the 5th Distributed Memory Computing Conference*, page to appear, Charleston, SC, April 9-12 1990.

[SS90] L. Snyder and D. G. Socha. An algorithm producing balanced partitionings of data arrays. In *Proceedings of the 5th Distributed Memory Computing Conference*, page to appear, Charleston, SC, April 9-12 1990.

[Stu77] M. Stutley. *Harper's Dictionary of Hinduism: Its Mythology, Folklore, Philosophy, Literature, and History.* Harper & Row, New York, NY, 1st U.S. edition edition, 1977.

[TGF87] S. Thakkar, P. Gifford, and G. Fielland. Balance: A shared memory multiprocessor. In *Proceedings of the Second International Conference on Supercomputing*, Santa Clara, CA, May 1987.

[TR88] L. W. Tucker and G. G. Robertson. Architecture and applications of the connection machine. *Computer*, 21(8):26–39, August 1988.

[Tse89] P. S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1989.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois, Urbana, IL, October 1982.

[ZBG88] H. Zima, H. Bast, and M. Gerndt. *Parallel Computing*, volume 6, chapter Superb: A Tool for Semi-Automatic MIMD/SIMD Parallelization, pages 1–18. North-Holland, Amsterdam, 1988.