

1994

Compilation techniques for irregular problems on parallel machines

Subhendu Das

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Das, Subhendu, "Compilation techniques for irregular problems on parallel machines" (1994).
Dissertations, Theses, and Masters Projects. William & Mary. Paper 1539623851.
<https://dx.doi.org/doi:10.21220/s2-vwj0-kj14>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9501413

**Compilation techniques for irregular problems on parallel
machines**

Das, Subhendu, Ph.D.

The College of William and Mary, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

COMPILATION TECHNIQUES FOR
IRREGULAR PROBLEMS ON PARALLEL MACHINES

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

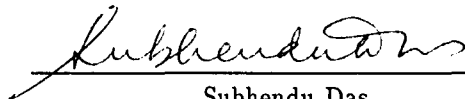
by

Subhendu Das

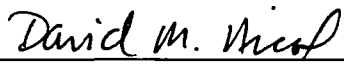
1994

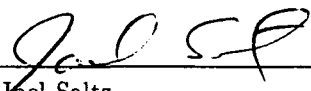
APPROVAL SHEET

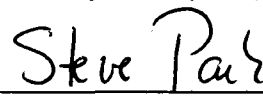
This dissertation is submitted in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy



Subhendu Das

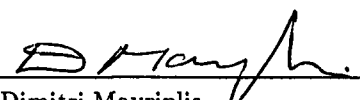
Approved, October 1993


David Nicol


Joel Saltz,
University of Maryland, College Park


Steve Park


Rahul Simha


Dimitri Mavriplis,
ICASE, NASA Langley Research Center

I dedicate this dissertation to my parents, Nisith and Kamal Das.

Contents

Acknowledgements	vii
List of Figures	viii
Abstract	x
1 Introduction	2
1.1 Parallel Programming	4
1.1.1 Parallel Architectures	4
1.1.2 Parallel Operating Systems	7
1.1.3 Parallel Languages and Compilers	8
1.2 Scientific Applications	9
1.2.1 Static Single Phase Computations	10
1.2.2 Multiphase Computations	11
1.2.3 Adaptive Irregular	11
1.3 Important Compiler Projects	13
2 Problem Definition and Approach	18
2.1 Parallelization of Irregular Loops	19
2.2 Partitioning Issues	20
2.2.1 Data Partitioning	21
2.2.2 Work Partitioning	22

2.3	Parallelization Schemes	22
2.3.1	Data Replicated Approach	22
2.3.2	Data Parallel Approach	23
2.4	Compiler Issues	26
2.5	Solutions Suggested in this Thesis	27
3	Compiler Support for Irregular Problems	29
3.1	Overview of HPF	30
3.2	Overview of the Initial PARTI Work	32
3.3	The PARTI Primitives	34
3.3.1	Paged Distributed Translation Table	36
3.3.2	Primitives for Generating Inspectors/Executors	42
3.3.3	Incremental Scheduling	49
4	Performance Analysis of Runtime Support	55
4.1	Applications Used for Performance Analysis	57
4.1.1	Real Applications	57
4.1.2	Synthetic Workload	59
4.2	Communication Optimizations	63
4.2.1	Software Caching	63
4.2.2	Communication Coalescing	65
4.2.3	Example Test Codes	67
4.3	Scaling Characteristics of the Optimizations	70
4.4	Experiments and Results	73
4.4.1	Synthetic Workload Performance Results	73
4.4.2	Performance Results Derived from Applications	77
5	Loop Transformations	83
5.1	Example Transformation	86

5.2	Preliminaries	92
5.3	Definitions	93
5.4	The Algorithm	96
5.4.1	Slice Graph Construction	96
5.4.2	Trace Management Schemes	99
5.4.3	Case 1: Low subscript reuse	100
5.4.4	Case 2: High subscript reuse	100
5.4.5	Code generation	101
5.4.6	Using Incremental Scheduling	103
6	Implementation Issues	104
6.1	Symbolic Analysis	104
6.2	Program Slicing	108
6.3	Program Slice Generation	111
6.4	Transformation Implementation	114
7	Conclusion and Future Work	115
7.1	Contributions of this Thesis	115
7.1.1	Development of Compiler Support	116
7.1.2	Compiler Transformation	116
7.2	Future Work	117

ACKNOWLEDGEMENTS

I cannot begin to thank Joel Saltz for his support, guidance and encouragement during the entire period of this research. I could always go to him for help, and his contributions during the development of the various concepts presented in this document are significant. Likewise, Dave Nicol was the only person who took a chance on this refugee from Mechanical Engineering and allowed me to pursue a graduate degree in computer science by hiring me as his research assistant. He also made significant contributions during the writing of this document. I would also like to thank the members of my committee for the time spent on my behalf.

Thanks goes to all the people and institutions who supported me financially; they include Joel, Dave, ICASE and various other government agencies. Also, I would like to thank ICASE/NASA LaRC, Caltech and NIH for allowing me use their various parallel machines.

Doing systems work needs much support, and stands on the work of many other graduate students and researchers. I would like to take this opportunity to thank the people involved in the Parascope project at Rice University for the use of their system, and especially to Reinhard von Hanxleden for all his help. Further I must thank Ravi Ponnusamy of Syracuse University for developing some of the partitioning "stuff" used in this thesis. Thanks are also due to Mustafa Uysal of University of Maryland for developing the neat synthetic workload generator. However, this list would not be complete without thanking Alan Sussman for the many fruitful discussions.

And finally, thanks to my parents and both my sisters for their constant encouragement and tolerance of my tantrums.

List of Figures

1.1	Unstructured Mesh	9
1.2	Static Single Phase Computation	10
1.3	Multiphase Computation	12
1.4	Adaptive Irregular Computation	13
2.1	Example of Simple Irregular loop	19
2.2	Example of Inspectors/Executors	25
3.1	Example of HPF Style Data Distribution	31
3.2	Sequential Code	35
3.3	Global Index Translation	38
3.4	Paged Translation Table (Replication = 0.0)	39
3.5	Paged Translation Table (Replication = 0.5)	40
3.6	Translation Table Functions	41
3.7	Inspector Code for Each Processor	43
3.8	Localize Mechanism	45
3.9	Parallelized Code for Each Processor	47
3.10	Incremental Schedule	50
3.11	Inspector Code for Each Processor Using Incremental Scheduling	51
3.12	Parallelized Code for Each Processor Using Incremental Scheduling	53
4.1	Simple Irregular Loop	57

4.2	Synthetic Workload Loops	62
4.3	Simple Communication Aggregation Case	68
4.4	Schedule Merging Case	69
4.5	Cost vs. volume of communication and Intersection Ratio (gather, $\mathcal{N}_{loop}=4$, P = 32, $\mathcal{R}_{int} = \text{IF}$)	74
4.6	Cost vs. volume of communication and Intersection ratio (Inspector, $\mathcal{N}_{loop}=4$, P = 32, $\mathcal{R}_{int} = \text{IF}$)	74
4.7	Cost vs Number of duplicates (Executor, low \mathcal{N}_{dup} , 32 Processors)	75
4.8	Cost vs Number of duplicates (Executor, high \mathcal{N}_{dup} , 32 Processors)	76
5.1	Kernel with single level of indirection.	85
5.2	CSR kernel – original version.	86
5.3	CSR kernel – transformed version (Part 1).	88
5.4	CSR kernel – transformed version (Part 2).	89
5.5	Slice graph generation algorithm.	97
5.6	Example of a Slice Graph.	99
5.7	Code generation algorithm.	102
6.1	Program fragment and SSA form	106
6.2	Code for Program Slicing	109
6.3	Slice for Slicing Criteria : $\langle S7, a \rangle$	110
6.4	Slice for Slicing Criteria : $\langle S8, b \rangle$	111
6.5	Program Dependence Graph for Slicing Example	112

ABSTRACT

Massively parallel computers have ushered in the era of teraflop computing. Even though large and powerful machines are being built, they are used by only a fraction of the computing community. The fundamental reason for this situation is that parallel machines are difficult to program. Development of compilers that automatically parallelize programs will greatly increase the use of these machines.

A large class of scientific problems can be categorized as irregular computations. In this class of computation, the data access patterns are known only at runtime, creating significant difficulties for a parallelizing compiler to generate efficient parallel codes. Some compilers with very limited abilities to parallelize simple irregular computations exist, but the methods used by these compilers fail for any non-trivial applications code.

This research presents development of compiler transformation techniques that can be used to effectively parallelize an important class of irregular programs. A central aim of these transformation techniques is to generate codes that aggressively prefetch data. Program slicing methods are used as a part of the code generation process. In this approach, a program written in a data-parallel language, such as HPF, is transformed so that it can be executed on a distributed memory machine. An efficient compiler runtime support system has been developed that performs data movement and software caching.

**COMPILATION TECHNIQUES FOR
IRREGULAR PROBLEMS ON PARALLEL MACHINES**

Chapter 1

Introduction

Techniques and methodologies have been developed that can be used to build compilers to parallelize scientific programs. Massively parallel computers have made the possibility of teraflop computing a reality. But programming a parallel machine is a non-trivial proposition. Two basic approaches exist for using a parallel machine. The first approach involves using a parallelizing compiler to generate parallel codes from sequential codes. The other approach consists of hand-parallelizing a given problem. Significant difficulties are associated with both these approaches. For any given program, a computational graph exists that needs to be mapped onto the target machine's topology. For programs written in C or Fortran, it is often very difficult for a parallelizing compiler to identify the underlying computational graph. In such cases, the process of automatic parallelization fails, and for that reason, very little success has resulted in generating parallel codes for real applications. When a code is hand parallelized, the user has to identify the computational graph and embed it in the machine's topology. The process of hand parallelization is very tedious and a certain amount of software has to be built for each code that is ported. Hand parallelization is not a very practical approach; therefore, a more automatic method of parallelization is desirable.

Automatic vectorization of scientific programs is accomplished aggressively by compilers. Vectorization can be done by recognizing the existence of certain vectorizable constructs in

the code. In the case of automatic parallelization, a similar approach is taken. Data parallel languages have been developed, which allow scientists to express the structure of problems accurately, thus allowing the compilers to do an efficient job. Researchers have successfully parallelized real application codes using data parallel languages, such as C*, *LISP and CM Fortran. Such languages have been fairly successful, and there is an effort to define a standard scientific data parallel language called High Performance Fortran (HPF).

The goal of effective parallelization of realistic applications is difficult to achieve. The author recognizes that the development of a single model and software support system to handle all types of applications is not feasible. Here, the computation domain is divided into broad classes, each of which is large enough to warrant separate software support. This development of specialized software models for each class of problem effectively captures the structure of the applications, thus helping in the generation of parallel codes by the parallelizing compilers. Tight coupling exists between the applications and software support. In general, applications can be broadly divided into two classes, namely, *regular* and *irregular* [42]. This classification is based on the underlying computational structure.

In this study, software support and compiler techniques have been developed that can be used to automatically parallelize irregular computations expressed in a data parallel language, such as HPF [56], Fortran D [40] or Vienna Fortran [107]. The software support developed here can also be used directly to parallelize irregular codes written in languages, such as Fortran 77 or C. The optimizations that have been incorporated into these software tools are targeted for distributed memory MIMD machines, like the Intel Gamma, Intel Delta and CM-5.

This chapter covers the necessary background required for understanding the research and also the relevant work present in the literature. Chapter 2 defines the problem and gives a high level description of the approach. In Chapter 3, software support developed to automatically parallelize irregular problems is presented. Chapter 4 presents the performance analysis done using the software tools. Chapter 5 presents the loop transformation algorithms that can be used by compilers to parallelize irregular applications. In Chapter 6,

the implementation details of the algorithms are presented, and in Chapter 7, conclusions and possible future work are discussed.

1.1 Parallel Programming

The requirement for huge amounts of processing power to solve large problems has forced the development of computer architectures that are different from the traditional von Neumann model. As problem size increases, designers have to move away from the computing model, wherein a single instruction is executed at a time to process a single datum. Concurrent computing, defined as several computers on a network working to solve a single large application, is an answer to this problem. The computers that participate in a concurrent computing environment may be identical to each other or each of them may have a different architecture.

Parallel programming is the branch of concurrent computing in which a collection of processors on a tightly coupled network cooperate to solve a large application. When a number of processors are taking part in a computation, it is likely that one processor will need some result calculated by another processor. If so, depending on the memory configuration, there might have to be explicit message passing between the processors and from time to time the processors might have to synchronize. Parallel programming raises a number of complicated issues depending on the type of parallel machine used.

1.1.1 Parallel Architectures

Parallel machine architectures can be broadly divided into two models, i.e., Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD). The different models are natural deviations from the von Neumann model of computing, which is often referred to as the Single Instruction Single Data (SISD) model. The parallel models can be further subdivided, based on the memory structure. The following memory structures exist:

Table 1.1: Examples of Parallel Machines

<i>Memory Model</i>	<i>SIMD</i>	<i>MIMD</i>
Shared	-	Cray Y-MP C916, Sequent, Convex 3880, NEC SX-3/44R
Distributed	Maspar MP-1, CM-2, CM-1, DAP	Intel Paragon, CM-5, MIT J-Machine
Distributed Shared	-	KSR-1, Stanford Dash

- Shared memory,
- Distributed memory,
- Shared distributed memory.

In the shared memory model, the group of processors that have been allocated to work on a particular problem has direct access to a single memory. Every data element is addressed via its global address. In the distributed memory model, each processor has exclusive access to its own particular chunk of memory. Data must be explicitly moved between different processor memories using message passing. Data elements are addressed using local addresses in each processor. In the shared distributed memory model, each processor has its own particular chunk of memory, like the distributed memory model, but each data element is addressed by its global address, using hardware support to automatically move data between processor memories. Hence, if a reference is made to a data element residing in another processor memory, the machine hardware moves the data element to local memory. The automatic movement of data elements between the processor memories requires considerably extra hardware support, and different protocols [69] are used to keep the memories coherent.

Single Instruction Multiple Data Model

In the SIMD model of computation, all processors execute in lock step. At a given time, every processor executes the same instruction but on a different piece of data. Synchronization is not a problem in such a model because either every processor wants a resource or none wants it. When all processors want a resource, the control processor broadcasts it to them. A single instruction memory exists in such a model and a single program counter. Examples of distributed memory SIMD machines are shown in Table 1.1. The recent trend has been to move away from SIMD architectures.

Multiple Instruction Multiple Data Model

The MIMD model of computation is more general than the SIMD model. In fact, if certain constraints were put on a MIMD computation model, one could achieve the SIMD model of computation. In the MIMD model, every processor executes a separate program. There is a separate program counter on each processor. Usually, the same program copy is executed on each processor (Single Program Multiple Data: SPMD), but the input data to each of the programs is different. Since all the processors work on the same problem, usually there are dependencies. If so, processors have to exchange data. The type of communication that takes place depends on the memory model of the machine.

In a shared memory MIMD machine, there is one large global memory visible to all processors. There is no explicit message passing in this case because processors communicate via shared variables. Different types of protocols, like *test-and-set*, *semaphores* and *fetch-and-add* are implemented to prevent deadlock. Shared memory bus-based architectures, such as the Sequent and Alliant, are not feasible for machines with a large number of processors because of the clear limitation of the single bus into the main memory.

In distributed memory MIMD machines, processors exchange data via message passing. A variety of different message passing protocols have been used [82, 100, 28]. Synchronization between processors can be achieved using message passing. Every processor has its own chunk of memory and only addresses its own memory. This architecture scales to

a large number of processors. The popular machines, these days, are MIMD distributed memory, e.g., the Intel Paragon, which involves a separate communications processor, associated with each computation processor (they all exist on a single board). In the future, four computation processors will share a single communication processor. The processors on a single board will have shared memory, but otherwise the memory will be distributed. The machine has a mesh communication structure. Another popular machine is the CM-5, which has a fat tree structure [17]. Each of the CM-5 nodes have a Sparc chip and four vector pipes. Both these machines have been successfully used to solve large scientific problems.

Another type of MIMD machine is the distributed shared memory machine. In this case, each processor has its own chunk of memory, but the processors address data in global name space. There is hardware support to automatically move data between local memories of processors. The MIT J-machine is a MIMD distributed memory machine, but the software makes it a distributed shared memory machine. The operating system, COSMOS, running on the J-machine, helps create the shared memory structure. The true distributed shared architectures, like the KSR-1 and the Stanford Dash machine, have to maintain elaborate directory structures for the purpose of cache coherency [68]. These machines are easier to use, compared to the fully distributed memory machines because of the existence of the global name space.

1.1.2 Parallel Operating Systems

Work on the development of parallel operating systems has been going on for the last two decades. The early operating systems, like Hydra [105], Medusa [79] and StarOS [59], were object-oriented and were developed for PDP-11 based machines. Hydra was designed for a shared memory architecture, and it allowed multiple processors to perform OS functions simultaneously. On the other hand, Medusa and StarOS were developed for a distributed memory machine. Both Medusa and StartOS were implemented as a collection of processes working with each other to solve a problem.

CMost is the operating system running on the CM-5 [28]. The computational processors on the CM-5 are grouped together to form *partitions*. The whole machine may operate as a single partition, and the smallest partition is comprised of 32 nodes. Each partition has a partition manager and runs the full CMost operating system, making all the allocation and swapping decisions. Each node in the partition runs a micro kernel that helps implement the CMost functions.

In the MIT J-Machine, the operating system, COSMOS [32], provides a global address space. It provides an *object* based memory management. Both data and codes are stored in objects and each object has a unique identification number. Objects can migrate between the nodes to reduce communication and provide load balancing. COSMOS provides the infrastructure required for fine-grained concurrent computation. Fast access to non-local memory is provided.

The operating system running on the DASH [69] machine was built by modifying the the Irix (Unix like) operating system. This operating system supports multiprogramming and multiple users on the system. The Irix operating system was changed to take advantage of the special features of the DASH architecture like pre-fetch, queue-based locks, etc.

There exists a large body of work in the literature dealing with distributed operating systems [73, 80, 12]. Most of the operating systems have been developed in the context of supporting a shared memory in a distributed memory machine. Since the context of this thesis is compilers and languages for parallel machines, the operating system issues are not discussed in detail. Only a brief overview of some of the current work is included here.

1.1.3 Parallel Languages and Compilers

Numerous projects in the field of parallel languages and compilers targeted for the various different architectures exist. Since parallel languages and compilers are closely related to the subject of the thesis, the literature review section (1.3) describes them in detail.

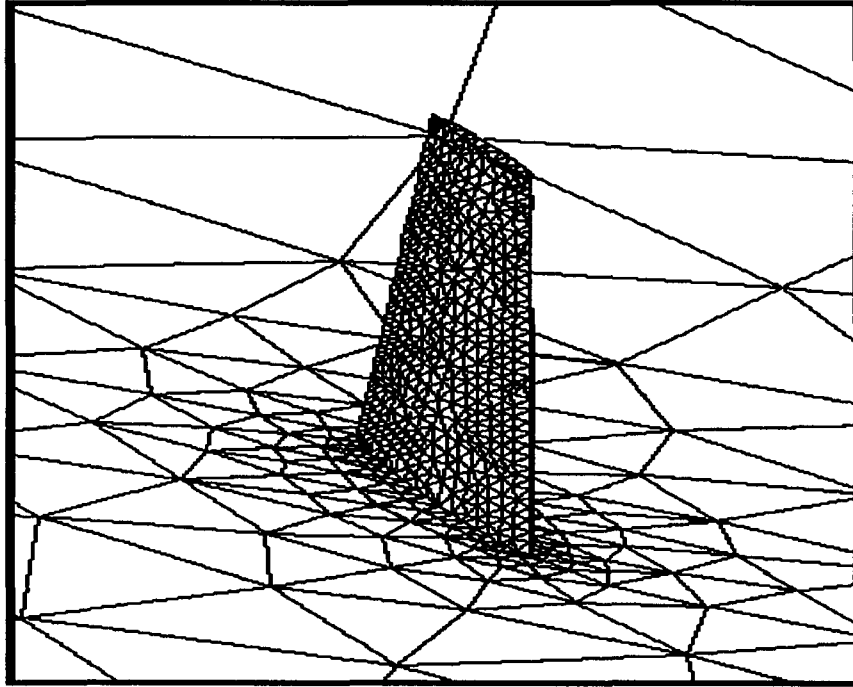


Figure 1.1: Unstructured Mesh

1.2 Scientific Applications

A detailed categorization of scientific applications is given by Fox [42, 27]. Applications are divided into the following four categories according to their temporal structure:

Synchronous: These applications are typically data-parallel with the time dependence calculation at each point on the computational graph done by the same operations. These problems are natural for parallelization on SIMD architecture.

Loosely Synchronous: These applications are also data-parallel, but the calculations performed at each point on the computational graph can be done by using separate algorithms. The points in the computational graph are often connected with each other in an irregular manner. Hence, these problems are often referred to as *irregular*. Arrays in irregular problems are typically indexed by *indirection arrays*. Figure 1.1 depicts an unstructured mesh.


```
L1 do j=1,timesteps  
  
  L2    do i=1,n_iterations  
  
    S2      n1 = ia(i)  
  
    S3      n2 = ib(i)  
  
    S4      y(n1) = y(n1) + x(n2)  
  
      end do  
  
  end do
```

Figure 1.2: Static Single Phase Computation

Asynchronous: These problems are irregular both in space and time. It is impossible to suggest a general method of parallelization of such problems. Each problem has to be parallelized separately.

Embarrassingly Parallel: All data points in the computational graph are disconnected both in space and time. These problems can be executed both on SIMD and MIMD architectures. Load balancing is the only consideration during partitioning.

This discourse will explore loosely synchronous problems in more detail, presenting examples from some of the application codes with which this researcher has worked.

1.2.1 Static Single Phase Computations

Loosely synchronous problems consist of concurrent computational phases that are repeatedly executed. The connectivity of the underlying computational graph does not change throughout the life of the computation. The piece of code shown in Figure 1.2 is an example of static single phase computation. The arrays **x** and **y** are indexed using the indirection arrays **ia** and **ib**. This type of computation is commonly found in unstructured mesh solvers. Examples of this type of computation consist of applications codes involving sparse matrix-

vector multiplications, explicit unstructured mesh solvers, etc. Efficient implementation of such problems consists of partitioning the data and the work so that communication is minimized and the load is balanced.

1.2.2 Multiphase Computations

Multiphase computations consider each phase as a static single phase computation with a specific computational graph. The solution from one phase of the computation is used to drive the solution in the next phase. Examples of multiphase computations are unstructured multigrid mesh solvers, particle-in-cell codes, etc. Partitioning these types of computation is very involved. Not only must the data and computation for a phase be partitioned, as if it was a static single phase computation, but the phase to which the data will be transferred after the computation has ended must be considered as well. The transfer of data between phases must be considered during partitioning since the results from one phase are used to drive the calculation in another. An example based on multigrid mesh solvers is presented in Figure 1.3. The example is a very simplistic representation of the type of computation that is required during multigrid solutions, but manages to portray the complexities involved. In the example, the arrays **wc** and **wf** store data values at the coarse and fine mesh points, respectively. There are two arrays, **Cweight** and **Fweight**, which are used to store the weights that are required during interpolation. The arrays **Cinter** and **Finter** are the interpolation arrays, required to transfer data between the various meshes. Calculation goes back and forth between the two phases, coarse and fine.

1.2.3 Adaptive Irregular

The example presented in Figure 1.4 depicts an adaptive irregular computation. After every timestep the computational graph changes, thus changing the indirection patterns. Rapid preprocessing is required to move data around. Data must be remapped to reduce communication volume, although it has been found that it is not required every timestep. Fast data partitioning algorithms are required to partition data before it can be remapped.

```
L1 do j=1,timesteps

  C Loop over Coarse mesh

  L2  do i=1,n_coarse

    S1  wc(i) = wc(i) + dwc(Cedge(i))

      end do

    C Interpolate from Coarse to Fine

    L3  do i=1,n_fine

      S2  wf(i) = wf(i) + Fweight(i) * wc(Finter(i))

        end do

      C Loop over Fine mesh

      L4  do i=1,n_fine

        S3  wf(i) = wf(i) + dwf(Fedge(i))

          end do

        C Interpolate from Fine to Coarse

        L5  do i=1,n_coarse

          S4  wc(i) = wc(i) + Cweight(i) * wf(Cinter(i))

            end do

        enddo

      enddo
```

Figure 1.3: Multiphase Computation

```
L1 do j = 1,timesteps  
  
L2   do i=1,n_iterations  
  
S1     y(i) = y(i) + x(edge(i))  
  
       end do  
  
C Computation to change the indirection array  
  
L3   do i=1,n_iterations  
  
S1     edge(i) = function(edge(i))  
  
       end do  
  
end do
```

Figure 1.4: Adaptive Irregular Computation

Examples of such computations are adaptive unstructured mesh solvers, molecular and particle dynamics codes, direct Monte Carlo simulations, etc.

1.3 Important Compiler Projects

Over the past few years, a considerable amount of work has been done in developing both shared and distributed memory compilers. In some approaches, parallel programming languages and environments have been developed, while in others a sequential language, like Fortran, is annotated so that transformations can be performed to generate parallel codes. In this section, a brief review of the important parallelizing compiler efforts may be found. First, the distributed memory compilers are reviewed, followed by a discussion of the shared memory compiler efforts.

Zima et al. developed the semi-automatic parallelization tool SUPERB [106, 44] for parallelization of programs for distributed memory machines. The SUPERB tool has an

interactive environment, and it transforms annotated Fortran programs into parallel codes. Initially, array element level communication statements are generated, after which aggressive message vectorization is performed using data dependency information. The compiler automatically generates array overlaps which are used to store off-processor data. Rectangular data distribution can be specified by the user to layout the data. For parameter passing between procedures interprocedural data-flow analysis is used.

Koelbel et al. [64, 61, 63, 62] designed the Kali compiler, the first to support both a regular and irregular data distribution. The development of the Kali language was based on BLAZE, a coarse-grained dataflow language [77]. The important parallel constructs in a program written for Kali are the data distribution statement, the virtual processor array declaration and the *forall* statement. The virtual processor array allows for the parameterization of the program, thus making it portable to various number of physical processors. All statements inside a *forall* loop can be executed in parallel. The iteration partition is accomplished by the special *on* clause. For irregular computation, an inspector/executor [78] strategy is used.

A distributed memory compiler, developed by Callahan and Kennedy, uses dependency analysis to perform transformations [21]. Like the SUPERB compiler, parallel code is generated from sequential Fortran with data decomposition statements. Various transformations are performed to optimize data movement.

DINO [90, 89, 91] is a parallel language developed to support distributed memory scientific computation. Unlike Kali, it uses an explicitly parallel model of computation and does not derive parallelism from the sequential code. The DINO language was developed based on the C programming language. Instead of Kali's processor array, a virtual parallel machine needs to be declared using the construct called an *environment*. The same data can be mapped to multiple environments, which can be mapped to a single physical processor. User-defined distributions are supported in DINO. There are no explicit communication statements, but nonlocal references are annotated by the user. When distributed arrays are passed as subroutine parameters, if necessary, array sections are communicated. Analysis for

message vectorization is not performed by the DINO compiler because the user annotates all non-local references. In an explicitly parallel language like DINO, it is fairly easy to express pipelined computation. DINO2 [87], an extension of the DINO language, has richer language support for writing parallel programs.

Chen et al. developed a functional programming language called Crystal [25, 72, 71] for programming distributed memory machines. The Crystal compiler does not have sophisticated dependency analysis tools; the existing dependencies are evident from the program text. Dependencies are analyzed to distribute the computation and the data. The central portion of Chen's work deals with automatic partitioning of data and work. The generation of the communication statements is done by subscript pattern matching [70]. The output from the Crystal compiler is a C program with message passing statements.

CMU Wrap [8] is a distributed-memory programmable systolic-array machine developed at Carnegie Mellon University. The language for this machine is AL and was developed by Tseng [98, 97]. Each cell of the systolic array machine is programmed using the language W2 [67]. The AL compiler generates W2 programs that can be executed on each of the cells. Data objects can be scalars, arrays or distributed arrays. Only a single dimension of an array can be distributed because the Warp machine is a linear array. A construct called *DO** is used to instruct the compiler to attempt parallelizing the Do-loop. The compiler does the parallelization if it can guarantee that the parallel order of the execution is same as the sequential order. The AL compiler does data and loop iteration partitioning based on *data relations*, which are those that exist between the different objects of the program. The compiler can handle general types of distribution. An automatic mapping compiler [96, 95] was developed for the applicative programming language, Sisal [76], where the target machine was the CMU Wrap. The mapping compiler applies different execution models to Sisal programs to determine the "best" mapping method. The execution models are developed based on the machine model and mapping models.

Rogers and Pingali developed the functional language Id Noveau [85, 84] to be used for distributed memory machines. They provide a single assignment array structure called

I-structures, which considerably simplify compiler analysis. Data mapping is done using functions supplied by the user. The user also provides the global to local address dereferencing functions. Communication and computation are pipelined using compile-time analysis. Runtime resolution of messages is performed and a separate node code is produced for each processor.

A C++ based language called C* [83], was developed by Quinn and Hatcher to support SIMD data parallel programs. The language C* was developed for the Connection Machine. In C* a virtual machine is declared. *Domains* that signify virtual processors are an abstract data type and are declared the same way classes are declared in C++. There is no global view of the data; all references must be made with respect to the local data structure. Data can be moved from one domain to another, and all communications are generated automatically. When a block of statements is specified to be executed on a domain (virtual processors), the statements are executed in parallel.

André et al. [6, 7] developed Pandore, where parallelism is extracted from the sequential code. The language also has constructs to explicitly express parallelism. Pandore can efficiently handle regular codes. Input to the Pandore compiler is an extended C program, wherein the user declares the parallel virtual machine, and the compiler automatically maps the data. For data communication, the compiler generates calls to the Pandore communication library.

The Aspar compiler, developed at Caltech by Fox et al. [57], takes sequential Fortran programs as input without annotations, and outputs a distributed memory code. The compiler must perform significantly more analysis than the other compilers described previously in order to perform the partitioning. For communication the Aspar compiler uses the Express [81] runtime primitives.

Fortran D [41, 45, 50, 53], developed at Rice University, is a parallel programming language based on Fortran 77, and can be used to write distributed memory programs. This language has added a rich set of extensions to Fortran to allow for data distribution. Fortran D supports irregular data mapping. The Fortran D compiler does a fair amount of

dependency analysis to figure out which loops can be executed in parallel. The compiler uses a two level mapping scheme, i.e., data is mapped onto a virtual processor array, which is then mapped onto the physical machine. All the loop transformations for irregular programs presented in this thesis have been implemented in the Fortran D environment.

Vienna Fortran [11, 23, 22] is a Fortran D like language developed for scientific computation. It does not have *decomposition* statements like those found in Fortran D. Vienna Fortran supports dynamic data decomposition as well as explicit processor array declarations. The Vienna Fortran compiler uses the PARTI primitives, described in this thesis, for irregular computation. A variety of other attributes can be specified for data distribution to deal with passing of distribution information between procedure boundaries.

A few of the important compilers that use data dependency analysis [66, 10] to generate parallel code for shared memory architectures are Parafrase [65], PTRAN [2] and PFC [3]. The compilers use standard Fortran input, recognize vector operations and compound functions, and reorganize code for execution on vector and parallel machines. The main goal is to extract the maximum amount of parallelism from the input code. A number of optimizations are performed by these compilers to obtain locality of memory reference. Improving the locality of memory reference makes good use of the registers and caches of a processor, thus boosting performance. The optimizations performed by the shared-memory compilers do not consider interprocessor data communication because of the presence of a global address space.

Some important parallel compiler projects have been covered in this section; however, there have been other compiler projects designed to support parallel computation [24, 74, 86, 43]. A few years ago, an effort was initiated to design a standard scientific parallel programming language; the result is High Performance Fortran (HPF) [56]. HPF is expected to be made available on most commercial machines. The first definition of this language does not support irregular mapping of data, but will be included in the revised definition.

Chapter 2

Problem Definition and Approach

This chapter presents an in-depth definition of the problem and gives a general outline of suggested solutions. Data parallel languages provide users with a wide range of constructs to distribute data and work between the processors of a distributed memory machine. Compilers written for such languages use data and work distribution information to generate efficient code to be executed on a parallel machine. The purpose of this dissertation is to

Define and develop compiler support and transformation techniques that can be utilized to automatically parallelize irregular problems, written in a data parallel language, to be executed on a distributed memory MIMD architecture.

The data parallel languages considered here are based on Fortran. The most basic and widely used construct in such languages is the **DO** loop. **DO** loops in which the data access pattern is irregular can be parallelized in a variety of ways. Each of the different methods has advantages and disadvantages. The parallelization method chosen depends on the architecture of the target machine. In this chapter, each of the methods will be described, with special emphasis on the method that was followed.

```
L1 do i=1,n_steps  
  
  L2 do j=1,n_edge  
  
    S1 n1 = nde(j,1)  
  
    S2 n2 = nde(j,2)  
  
    S3 flux = F(x(n1),x(n2))  
  
    S4 y(n1) = y(n1) + flux  
  
    S5 y(n2) = y(n2) + flux  
  
  end do  
  
end do
```

Figure 2.1: Example of Simple Irregular loop

2.1 Parallelization of Irregular Loops

In irregular loops data arrays are indexed using indirection arrays. Therefore, the access patterns are known only at runtime, after the indirection arrays are initialized. An example of a simple irregular loop is presented in Figure 2.1. This example will be used to present the different methods to parallelize irregular loops. The code shown in Figure 2.1 is a simplified version of loops extracted from a real computational fluid dynamics code. This illustration involves looping through the edges of an unstructured mesh and calculating the flux. The outside loop is executed for `n_steps`, which is usually an input parameter. The parameter is chosen depending on some convergence criteria. The indirection array `nde` is two-dimensional in structure, where `nde(j,1)` and `nde(j,2)` are the two nodes in the unstructured grid connected by edge `j`. The two data arrays are `x` and `y`. For each iteration the value of the variable `flux` is calculated using `x(n1)` and `x(n2)`. The calculated flux is stored in array `y`.

The primary objective is to execute the loops shown in Figure 2.1 on a distributed memory parallel machine. The purpose is to have a parallelizing compiler generate the required code. The compiler has to perform two steps: The first step is to partition the data and the work between the processors; the next step is to generate a code that each processor can execute. There must be sufficient input from the user to the parallelizing compiler to achieve these objectives. Issues involved in generating the parallel code are described in this chapter.

2.2 Partitioning Issues

To parallelize the loops shown in Figure 2.1 so that they can be executed on a distributed memory machine, both the data and the work must be distributed to the different processors. After the data and work are distributed among the participating processors, each processor executes the loop nest. The outer loop remains unchanged; the inner loop bounds are changed to the number of local iterations assigned to each processor. For the example code shown in Figure 2.1, the number of local iterations of the inner loop assigned to each processor is determined by the number of unstructured mesh edges assigned to each processor. When each processor executes the iterations assigned to it, references may be made to non-local data. In such cases data communication between processors has to take place for successful completion of the computation. The data partitioning and the work partitioning are very much coupled. During the partitioning of both the data and the work, careful consideration is taken to keep the data communication between processors to a minimum. The amount of work distributed among the processors is maintained more or less equal so that the load is balanced. In most data parallel languages developed for parallel programming, there exists some sort of construct used to specify how to partition both the data and the work. Hence, the user controls the partitioning process.

2.2.1 Data Partitioning

Depending on the nature of the problem, the user can choose a number of different partitioning strategies. The partitioning scheme is chosen to reduce data communication between processors. The most commonly used partitioning schemes are the following:

BLOCK: In block distribution, an equal number of contiguous elements of an array are allocated to the processors, assuming the total number of elements is divisible by the number of processors. Hence, if there are n elements in an array, and there are p processors, assuming that $n \bmod p$ is 0, then each processor gets $n \div p$ elements with processor 0 getting the first portion, processor 1 the next portion and so on. If $n \bmod p$ is non-zero, some pre-defined strategy may be used to distribute the extra elements. BLOCK distribution is very common and used by most regular applications.

CYCLIC: In cyclic distribution, instead of allocating contiguous portions to the processors, each element is given to the processors one at a time and wrapping around is performed whenever necessary. Again, if there are n elements in an array, and there are p processors, assuming $n > p$, Processor 0 gets the first element, Processor 1 gets the second and so on. Since $n > p$, the allocation wraps around, i.e., Processor 0 gets element $p + 1$, Processor 1 gets element $p + 2$ and so on. CYCLIC distribution is less common than BLOCK distribution, but it is used in certain types of regular problems.

Irregular: Irregular distribution is commonly used for irregular problems. Here the actual data distribution is specified by a *map array*, which is the same size as the data array that needs to be distributed and contains processor numbers. Hence, $map(i)$ specifies the processor to which the data element i needs to be allocated.

There are other partitioning strategies like **BLOCK-CYCLIC**, **Irregular-Block**, etc. For multi-dimensional arrays, usually one dimension is distributed and the other dimensions are *compressed*. Compressing a dimension means that it remains undistributed.

2.2.2 Work Partitioning

Work distribution is performed by partitioning the loop iterations. The partitioning of loop iterations is very much dependent on the data distribution. Work is partitioned to reduce the inter-processor data communication and to balance the load. A few of the common schemes for performing work partitioning are presented.

Owner Computes: In this scheme, a particular iteration of a **DO** loop is allocated to the processor that contains the left-hand side of the statement, i.e., the element that is being updated. This partitioning is the most commonly used scheme, and is followed as closely as possible in this thesis. In irregular problems, “owner computes” does not always provide the best result.

On Home: Most data parallel languages allow **DO** loops to be annotated using the “on home” clause. This directive is used to assign iterations to different processors. Iterations can be assigned to processors containing either the right- or left-hand side of a statement.

A number of compilers have implemented the “owner computes” scheme successfully for regular problems. Other work partitioning schemes have been used where loop iteration is assigned, based on the ownership of the maximum number of data references.

2.3 Parallelization Schemes

In this section, some of the different strategies that can be used to parallelize the loop shown in Figure 2.1 will be outlined. Any of these strategies can be utilized, depending on the architecture and available resources.

2.3.1 Data Replicated Approach

In the data replicated approach, data is not partitioned between the participating processors but is replicated on all processors. The work, on the other hand, is partitioned between the

processors in such a way that the load is balanced, making the parallelization process very simple, uninteresting and communication intensive. At certain points in the code, every processor communicates with all others so that the data on each processor are identical.

The parallel version of the loop shown in Figure 2.1 has the same structure as the sequential loop, except that the loop bounds on each processor are different. After each processor executes the iterations that have been allocated to it, the processors communicate with each other so that the *y* array values on each processor are the same.

2.3.2 Data Parallel Approach

The data parallel approach of parallelization involves partitioning of both the data and the work. Depending on the architecture of the target machine, this category can be further subdivided. Most of the new parallel languages being developed are intended for data parallel programming. A number of interesting synchronization issues are raised when this approach is taken.

Inspectors/Executors

Using inspector/executor is the natural way to parallelize an irregular loop [78]. An irregular loop is transformed into two constructs, the inspector and the executor. The inspector is a piece of parallel software that at runtime analyzes the indirection arrays of an irregular loop and figures out the data access pattern. Once the data access pattern is generated, the inspectors, running on different processors, communicate with each other to determine the send/receive patterns of the data. These patterns are stored in a data structure called the *schedule*. A schedule needs to be generated for each type of data access pattern. The executor is the code that is executed to solve the problem. In iterative methods, it is frequently the case that a loop's inspector is executed once, while its executor is executed many times. The inspector/executor method of parallelization works well for loops with just *output dependencies* [5] as the one shown in Figure 2.1. When loops have dependencies that are *loop-carried* [5] then the inspector/executor method of parallelization

does not work well.

After the schedules required in an irregular loop have been generated by the inspector, the executor phase begins. During the executor phase, the already generated schedules are used to fetch the actual data and the off-processor data are stored in buffers. Once the data has been fetched, the actual computation can begin (the actual computation is also part of the executor). The pre-fetching of the data causes an overall reduction in the time required to execute the loop by reducing both the number of startups and the communication volume. If the same off-processor data are accessed more than once, only a single copy is fetched.

Figure 2.2 shows the parallel version of the loop depicted in Figure 2.1, using the inspector/executor parallelization strategy (this transformation is generated by a source-to-source translator). The loop structure in the executor is the same as the loop structure in the sequential code. The indirection array `nde`, which is in global coordinates, has been changed to local (i.e., processor) coordinates and is called `nde_local`. The loop bounds have been changed to the number of local iterations. The executor on each processor communicates with the participating processors at two points. The first communication point occurs before the execution of the actual computation. All data that will be used inside the loop are pre-fetched. After the actual computation, off-processor data are accumulated through another phase of communication. Both of these communication phases utilize the schedule that was generated in the inspector phase.

Fetch on Demand

At the beginning of every iteration a check is performed to determine data ownership. The processors that have non-local data references initiate fetches and block until the data arrives. An interrupt-driven message passing mechanism is required for the fetch on demand type of data transfer to work efficiently. Hardware or software support [100] is required for interrupt-driven message passing mechanism. The fetch on demand mechanism can only work on machines with very low latencies. The advantage of this method is one does not pay for the generation of the inspector. The fetch on demand method can be further optimized

S1 Inspector code analyzes local nde and generates schedule

Executor starts here

L1 do i=1,n_steps

S2 Using the schedule from the inspector gather x

L2 do j=1,n_edge_local

S3 n1 = nde_local(j,1)

S4 n2 = nde_local(j,2)

S5 flux = F(x(n1),x(n2))

S6 y(n1) = y(n1) + flux

S7 y(n2) = y(n2) + flux

end do

S8 Using the schedule from the inspector accumulate off-processor y

end do

Executor ends here

Figure 2.2: Example of Inspectors/Executors

by storing off-processor data in a special buffer that can be consulted before issuing new fetch initiations. Fetch on demand with buffering potentially reduces the number of off-processor fetches.

2.4 Compiler Issues

The main objective of this research is to answer the compiler issues raised when one wants to automatically parallelize irregular loops. This section presents a high-level description of the various types of analyses required to automatically generate inspector/executor pairs.

The compiler for a data-parallel language must first analyze the data and work distribution directives given by the programmer. The distribution information is required for generation of both the inspector and the executor. If two data arrays are indexed by the same indirection array but have different distributions, separate schedules for data communication must be built for each array. On the other hand, if the data arrays are distributed identically, then one schedule will suffice. The work distribution statements are utilized to generate the loop bounds.

The compiler takes a data-parallel program written in global coordinates as input and transforms it so that it can be executed on the node of a parallel machine. The SPMD (Single Program Multiple Data) model of computation is followed. The compiler splits the irregular loops and generates the inspector and executor codes.

The inspector generation involves three phases:

- Finding the indirection arrays used in a loop.
- Analyzing the references that use indirection arrays and decide what schedules should be generated. Analysis is done so that multiple copies of the same schedule is not generated.
- Performing optimization so that duplicate copies of the same off-processor data will not be fetched during the executor phase. This optimization has to be done during

the inspector building phase since, during the executor run, the information stored inside the schedule is used directly without further analysis.

Software primitives that generate data communication schedules when invoked with indirection arrays have been developed. To generate inspectors, the compiler inserts calls to these primitives which, during runtime, generate the required schedules.

Before generation of the executor code, the compiler must determine:

- The data references that require off-processor fetches and the schedules that will be generated during runtime in the inspector phase for these data references. Based on this information, the compiler makes calls to the gather/scatter routines.
- Whether more than one schedule must be used to access data from the same array.

In such cases the communication calls can be merged.

The optimizations to reduce the volume of data communication and the number of message startups have been built into the tools developed for generation of inspectors and executors. Also, transformation strategies have been developed to further optimize the volume of communication.

2.5 Solutions Suggested in this Thesis

This research makes concrete contributions in the area of automatic parallelization of irregular codes. In this thesis, the necessary requirements to build a parallelizing compiler for irregular problems have been presented. Also, the solutions suggested have been implemented in the prototype compiler.

An efficient compiler runtime support system that performs data movement between processors and software caching has been developed. The system is a portable library that can be used by any parallelizing compiler. Numerous compilers use this runtime system [18, 16].

Detailed transformation techniques for irregular computations that can be used by a compiler to generate parallel codes have been presented. These transformation methods are developed based on *program slicing* [103] techniques. Using these transformations, efficient parallel codes can be generated by a compiler for irregular problems written in data-parallel languages.

Chapter 3

Compiler Support for Irregular Problems

This chapter focuses on the design of a suite of tools that has been developed to support the transformation of irregular programs that run on uni-processors to ones that can be executed on parallel machines. The tools can be used by compilers or by users directly to generate parallel codes. The tools have been used to implement a number of large irregular codes on distributed memory machines. This author's prototype compiler has also utilized these tools to parallelize irregular loops extracted from various codes.

This author has developed methods necessary to generate efficient distributed memory codes for a large class of sparse and unstructured problems. In these problems, the dependency structure is determined by variable values known only at runtime. In such cases, effective use of distributed memory architectures is made possible by a runtime preprocessing phase, which is used to partition work, map data structures, and schedule the movement of data between the processor memories. The code required to carry out runtime preprocessing can be generated by a distributed memory compiler during a process called *runtime compilation* [93].

Once data structure and loop iteration partitioning have been determined, further preprocessing is carried out to generate communication calls needed to efficiently transport

data between processors. In sparse and unstructured computations, distributed arrays are typically accessed using indirection arrays. Runtime preprocessing is used to generate a small number of communications calls to carry out the required data transport. In many cases several loops access the same off-processor memory locations. As long as it is known that the values assigned to off-processor memory locations remain unmodified, it is possible to reuse stored off-processor data. A mixture of compile-time and run-time analysis can be used to recognize such situations. Compiler analysis determines when it is safe to assume that the off-processor data are valid. Software primitives generate communications calls that selectively fetch only those off-processor data, not available locally.

3.1 Overview of HPF

This section involves an overview of a data-parallel language that has been developed to support scientific computations. Researchers from both industry and academia established a forum to design a data-parallel language, High Performance Fortran (HPF) [56], that can be used to write scientific programs for both SIMD and MIMD architectures. The starting point for HPF was Fortran 90 because of its dynamic allocation and array operation features. Other features that have been added to HPF are based on the numerous parallel languages developed both by computer scientists and applications engineers. The data distribution directives added to the language are based on the extensions defined in Fortran D [40], and Vienna Fortran [107]. Some of the important features of HPF are presented here.

Constructs are present in this language that allow the programmer to explicitly specify parallel execution. The `INDEPENDENT` directive precedes a loop; its purpose is to tell the compiler that the statements in the loop do not have any sequentializing dependencies, and that they can be executed in any order without changing the semantics of the program. The `INDEPENDENT` directive allows the compiler to make various decisions regarding data placement and optimizations. The `FORALL` executable construct in the language allows simultaneous assignment of a large number of array elements. Fortran 90 has a `FORALL`

```
....  
  
S1      REAL *8 x(L,N), y(M,N), z(N)  
  
S2 !HPF PROCESSORS P(10)  
  
S3 !HPF DISTRIBUTE z(BLOCK) ONTO P  
  
S4 !HPF ALIGN (*,:) WITH z:: x, y  
  
....
```

Figure 3.1: Example of HPF Style Data Distribution

statement in its definition. HPF relaxes many of the restrictions regarding array assignments in the Fortran 90 `FORALL` statement.

Distributing data between the different processor memories is a very important aspect of parallel programming. Any data-parallel language should have constructs by which the user can specify the required data decomposition. Many researchers have explored the problem associated with specifying data decomposition [104, 88, 30, 83, 26, 71, 70]. The data distribution features of HPF allow the programmer to distribute data so that locality of data is maintained on each processor, thereby reducing data communication time. The `DISTRIBUTE` directive is used to specify how the data is to be mapped to an arrangement of virtual processors. At the moment, only *regular* distributions are defined in the HPF language, and multiple dimensions of an array can be distributed. Regular distributions that are supported in the language are `BLOCK` and `CYCLIC`. When data is distributed regularly, the address of any data element can be found by using an algebraic expression involving the processor number and distribution size. Data arrays can be redistributed by using the directive `REDISTRIBUTE`. The other data distribution directive, `ALIGN`, is used to group data objects that are to be distributed identically. Alignment of objects can be either static or dynamic.

An example of HPF style data distribution is depicted in Figure 3.1. There are three distributed arrays, **x**, **y** and **z**; **x** and **y** are two-dimensional arrays, while **z** is one-dimensional. The statement S2 declares a set of virtual processors, **P**, in the shape of a linear array. The statement S3 specifies that **z** should be block distributed across the set of virtual processors. The statement S4 aligns **x** and **y** with **z**. The align statement states how the two dimensions of **x** and **y** are to be distributed. The first dimension of the arrays **x** and **y** is to be *collapsed* onto the set of virtual processors, i.e., the first dimension is not distributed. The character “*”, shown in the first dimension in the align statement, signifies that the first dimension of **x** and **y** is to be collapsed. The second dimension of the arrays **x** and **y** is distributed, conforming to the distribution of the array **z**. The character “:”, shown in the second dimension in the align statement, signifies that the second dimension of **x** and **y** is to be aligned with the distribution of **z**.

HPF has been developed to be machine independent. For instance, the user may want to do explicit operations based on the architecture on which the program will be executed. HPF allows the program to call extrinsic procedures containing user defined machine dependent operations. Extrinsic procedures constitute one way to declare and accomplish operations on local data otherwise impossible to define within the context of the language.

The features of HPF that the author uses in some examples have been presented here. The complete language specification is presented in High Performance Fortran Language Specifications [56].

3.2 Overview of the Initial PARTI Work

The work thus far has been developed based on the initial development of a suite of primitives for implementing irregular problems on distributed memory parallel architectures. These primitives are named PARTI (Parallel Automated Runtime Toolkit at ICASE) [13, 93]. In this section an overview of the functionality of the PARTI primitives is given. In many algorithms, data produced or input during a program’s initialization play a large role

in determining the nature of the subsequent computation. In the PARTI approach, when the data structures that define a computation have been initialized, a preprocessing phase follows. Vital elements of the strategy used by the rest of the algorithm are determined by the preprocessing phase.

In distributed memory MIMD architecture, there is typically a non-trivial communications latency or startup cost. For efficiency, information to be transmitted should be collected into relatively large messages. The cost of fetching array elements can be reduced by precomputing the data each processor needs to send and receive.

In irregular problems, such as solving PDEs on unstructured meshes and sparse matrix algorithms, the communication pattern depends on the input data. The dependency on input data typically arises due to some level of indirection in the code. In such cases, it is not possible to predict at compile time what data must be prefetched. To deal with this lack of information, the original sequential loop is broken up into the inspector/executor pair. A brief description was given in the previous chapter of the inspector/executor construct; a more detailed description of this type of transformation is given in Mirchandaney et al. [78].

During program execution, the inspector examines the data references made by a processor and calculates what off-processor data need to be fetched and where the data will be stored once received. Inspectors on separate processors coordinate this task. The executor loop uses the information from the inspector to implement the actual computation. PARTI primitives can be used directly by programmers to generate inspector/executor pairs.

PARTI primitives carry out the distribution and retrieval of globally indexed, but irregularly distributed, data-sets over the numerous local processor memories. Each inspector produces a set of *schedules*, specifying the communication calls needed to either

- (i) obtain copies of data stored in specified off-processor memory locations (i.e., *gather*),
or
- (ii) modify the contents of specified off-processor memory locations (i.e., *scatter*), or
- (iii) accumulate (e.g., add or multiply) values to specified off-processor memory locations

(i.e., *accumulate*).

In distributed memory machines, large data arrays need to be partitioned between local memories of processors. The partitioned data arrays are called *distributed arrays*. Long-term storage of distributed array data is assigned to specific memory locations in the distributed machine. Frequently, partitioning distributed arrays in an irregular manner is advantageous. For instance, the numbering of the nodes of an irregular computational mesh does not have a useful correspondence to the connectivity pattern of the mesh. The data structure in such problems is partitioned to reduce interprocessor communication. This may cause the assignment of arbitrary array elements to each processor (*irregular partitioning*).

Each element of a distributed array is assigned to a particular processor. When an array is partitioned irregularly, finding the address of a particular data element of that array is a non-trivial task. Since any data element can reside in any processor, a global mapping table is set up to store the address information. This mapping table is called the *translation table*, and for each element of the data array, it stores the processor where the data reside and the local address in the processor.

3.3 The PARTI Primitives

In this section the primitives that have been developed for the generation of inspector and executor constructs, starting from sequential irregular loops, are described in detail. Primitives schedule and carry out movement of data between the processor memories. Various optimizations are performed aggressively to reduce data communication volume and message startups. The primitives have been designed to

- (i) eliminate redundant off-processor references, and
- (ii) simplify producing parallelized loops that are virtually identical in form to the original sequential loops.

A paged distributed translation table has been developed to reduce the time required to do address translation for irregularly distributed data.

```
real*8 x(N), y(N)

C Loop over edges involving x, y

L1 do i=1 , n_edge

    n1 = edge_list(i)

    n2 = edge_list(n_edge + i)

S1   y(n1) = y(n1) + x(n1) + x(n2)

S2   y(n2) = y(n2) + x(n1) + x(n2)

    end do

C Loop over Boundary faces involving x, y

L2 do i=1,n_face

    m1 = face_list(i)

    m2 = face_list(n_face + i)

    m3 = face_list(2 * n_face + i )

S3   y(m1) = y(m1) + x(m1) + x(m2) + x(m3)

S4   y(m2) = y(m2) + x(m1) + x(m2) + x(m3)

    end do
```

Figure 3.2: Sequential Code

To explain how the primitives work, an example, similar to loops found in unstructured computational fluid dynamics (CFD) codes, is used. In most unstructured CFD codes, a mesh is constructed that describes an object and the physical region in which a fluid interacts with the object. Loops in fluid flow solvers sweep over the mesh structure. The two loops shown in Figure 3.2 represent a sweep over the edges of an unstructured mesh followed by a sweep over faces that define the boundary of the object. Since the mesh is unstructured, an indirection array is used to access the vertices during a loop over the edges or the boundary faces. In loop L1, a sweep is carried out over the edges of the mesh and the reference pattern is specified by an integer array `edge_list`. Loop L2 represents a sweep over boundary faces, and the reference pattern is specified `face_list`. The array `x` only appears in the right-hand side of the expressions in Figure 3.2, statements S1 through S4, so the values of `x` are not modified by these loops. In Figure 3.2, data are read from, and written to, array `y`. These references involve accumulations in which computed quantities are added to specified elements of `y` (statements S1, S2, S3 and S4).

3.3.1 Paged Distributed Translation Table

When irregular problems are solved on distributed memory parallel machines, it is frequently advantageous to partition the data arrays irregularly. Data structures are partitioned to minimize interprocessor communication, and the partitioning may lead to arbitrary assignment of array elements to each of the processors. Once distributed arrays have been partitioned between processors, each processor ends up with a set of globally indexed distributed array elements that will be accessed during the executor phase.

Each element in a size S distributed array, A , is assigned to a particular home processor. In order for another processor to be able to access a given element, $A(i)$, of the distributed array the home processor where $A(i)$ resides must be known; also, the local address of $A(i)$ must be known. A *translation table* is built, that for each array element lists the home processor and the local address in the home processor's memory.

Memory considerations make it clear that it is not always feasible to place a copy of

the translation table on each processor. A translation table can be distributed between processors. Earlier versions of PARTI supported a translation table that was partitioned between processors in a blocked fashion [35], [104]. The partitioning was accomplished by putting the first N/P elements on the first processor, the second N/P elements of the table on the second processor, etc., where P is the number of processors. If access is required to an element $A(m)$ of distributed array A , the home processor and local offset for $A(m)$ is found in the portion of the distributed translation table stored in processor $((m-1)/N)*P+1$. A translation table lookup aimed at discovering the home processor and the offset associated with a global distributed array index is referred to as a *dereference request*.

In many cases, the naive translation table described above tends to be costly to use because

- the distribution of the translation table between processors is fixed and bears no particular relationship to the distribution of dereference requests; and
- some distributed array elements are included in a number of reference requests. In many cases, there is enough memory to partially (or completely) replicate the translation table. The naive distributed translation table is not able to replicate portions of the translation table in order to trade memory for improved performance.

In this section, a *paged* translation table is discussed. The translation table is decomposed into fixed-sized pages which list the home processors and offsets associated with a set of B contiguously numbered distributed array indices. Each processor stores $(P * a)$ pages, and at least one processor maintains a copy of each page; consequently, the total number of stored pages $(P * P * a)$ must be greater than or equal to the distributed array size S divided by B . Following the convention in the virtual memory literature, the memory location associated with each page is called a *page frame*. Each processor maintains a complete page table; for each page, the page table lists a processor and a page frame.

Translation table information for each index must be stored somewhere, simplifying the assumption that each processor must store at least $S/(B * P)$ pages. In the current paged

Dereference :

Translate local list of global indices to a list of (processor, offset) pairs

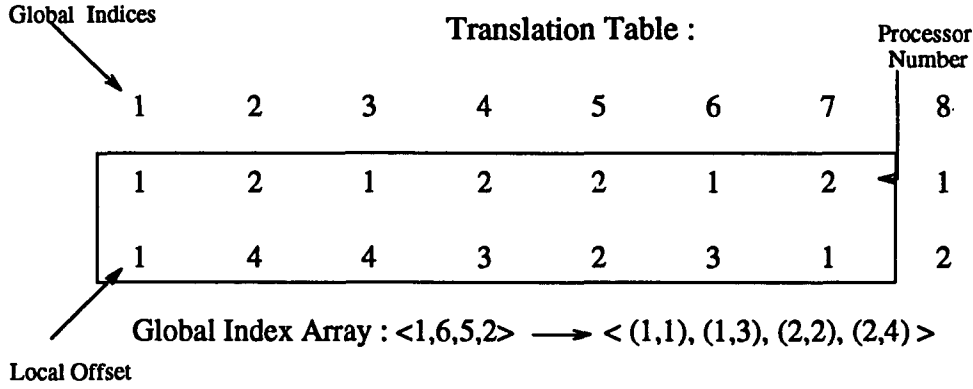
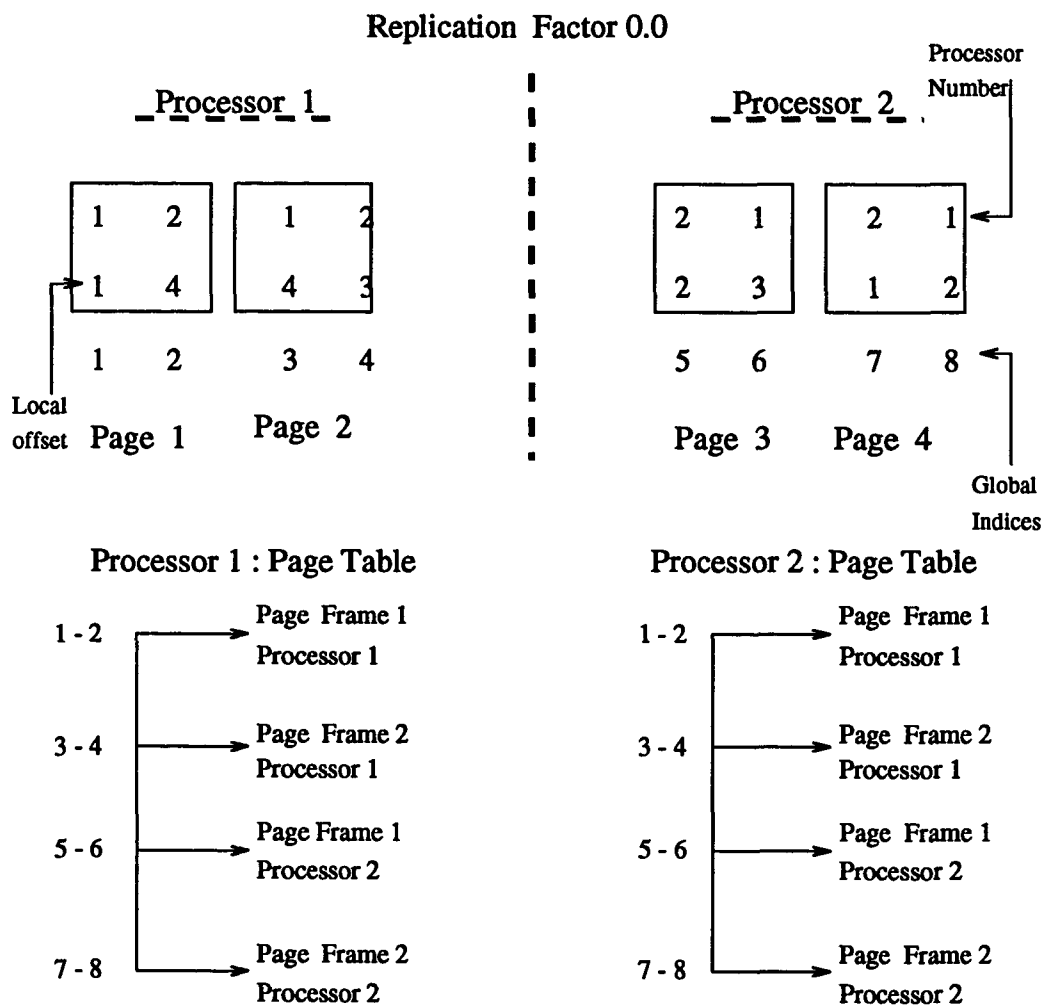


Figure 3.3: Global Index Translation

translation table implementation, $S/(B * P)$ pages are statically bound to each processor and copies of additional pages are dynamically assigned to each processor. In the absence of any memory constraints, each processor could dynamically store $S * (P - 1)/(B * P)$ pages; in this case, the entire translation table would be replicated. The *replication factor* (RF) is defined as the fraction of the maximum number of pages for which frames are allocated by each processor. The user (or compiler) sets the page size B and a *replication factor* (RF). Figure 3.3 shows the index translation process. Figure 3.4 depicts a highly simplified scenario in which there are 2 processors, an 8 element distributed array ($S=8$), a page size of 2 ($B=2$) and a replication factor of 0.0 ($RF=0.0$). Since no pages are replicated each processor has the same page table. In Figure 3.5 a scenario that is identical to the one, shown in Figure 3.3 is depicted, except now the replication factor is changed to 0.5. In this case, processor 1 contains a dynamic copy of page 3, and processor 2 contains a dynamic copy of page 1.

The runtime support allows each processor to choose which pages to replicate, based on the characteristics of a user (or compiler) specified distributed array access pattern, specified by integer array IA . Each index i of IA is dereferenced by consulting page $\frac{IA(i)-1}{B} + 1$. On each processor, the most heavily accessed pages are chosen as the dynamically



$$\text{Position in Table} = ((\text{index} - 1)/B) + 1$$

$$\text{Position in Page} = (((\text{index} - 1)\%B) + 1)$$

Figure 3.4: Paged Translation Table (Replication = 0.0)

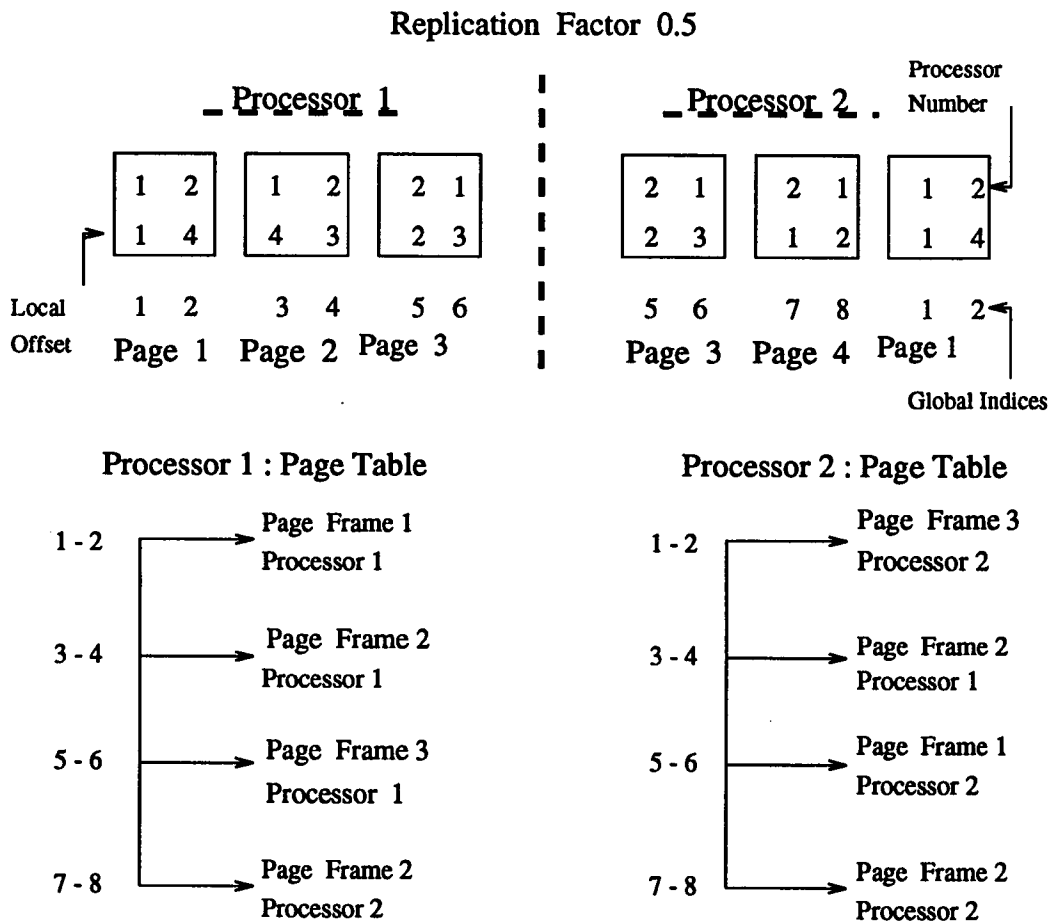


Figure 3.5: Paged Translation Table (Replication = 0.5)

<p>S1 <code>translation_table = BuildDsTable(myvals, on_proc, replication_factor)</code></p> <p>S2 <code>call DsShuffle(translation_table, index_array, ndata)</code></p> <p>S3 <code>call DerefDsTable(translation_table, index_array, local, proc, ndata)</code></p>

Figure 3.6: Translation Table Functions

assigned ones.

Translation Table Generation

The different function calls, used to generate and use the paged distributed translation table, are shown in Figure 3.6. The numbering of the nodes of an irregular mesh frequently does not have a useful correspondence to the connectivity pattern of the mesh. When such a mesh is partitioned in a way that minimizes interprocessor communication, it may be necessary to be able to assign arbitrary mesh points to each processor. The PARTI procedure *BuildDsTable* (S1 in Figure 3.6) allows storage of the mapping of a globally indexed distributed array in a regular (replicated or partially replicated) fashion.

On each processor the function *BuildDsTable* is passed:

1. A list of the array elements for which it will be responsible (*myvals* in S1, Figure 3.6).
2. The number of array elements for which this particular processor is responsible.
3. The percentage of the total translation table that is replicated on each processor. It is specified by the *replication_factor* in S1 shown in Figure 3.6.

The function *BuildDsTable* returns a pointer to the translation table. If a given processor needs to obtain a datum that corresponds to a particular global index i for a specific distributed array, the processor can consult the paged distributed translation table to find the datum's location in the distributed memory.

The PARTI call *DsShuffle*, shown in statement S2 in Figure 3.6, is used to move the pages of the translation table. Pages of the translation table can be shuffled to improve the locality

of addresses during the dereferencing phase. The shuffling capability of the translation table becomes important when parallelizing adaptive problems (frequent dereferencing is required).

On each processor the function *DsShuffle* is passed:

1. A pointer to the translation table whose pages are being shuffled.
2. The `index_array` according to which the pages are shuffled.
3. The number of elements in the `index_array`.

The function *DsShuffle* returns the modified translation table pointer.

The PARTI function *DerefDsTable* is used to obtain the addresses of distributed elements. On each processor the function *DerefDsTable* is passed:

1. A pointer to the translation table to be used for dereferencing.
2. The global indices, `index_array`, for which the local addresses are required.
3. The total number of elements `ndata`, for which dereferencing is required.

The function returns:

1. A processor list which is the same size as `index_array`.
2. A local offset list which is the same size as `index_array`.

The functions presented in this section can be used to build and access the translation table.

3.3.2 Primitives for Generating Inspectors/Executors

In this section, the primitives used to generate inspectors and executors are presented. The inspector code for the loops shown in Figure 3.2 is illustrated in Figure 3.7, and the corresponding executor code is shown in Figure 3.9.

```

S1 translation_table = BuildDsTable(myvals, on_proc, replication_factor)

S2 call localize(translation_table, edge_sched, part_edge_list, local_edge_list,
                2 * local_n_edge, edge_off_proc)

S3 call localize(translation_table, face1_sched, part_face_list, local_face_list,
                2 * local_n_face, face1_off_proc)

S4 call localize(translation_table, face2_sched, part_face_list(2 * local_n_face + 1),
                local_face_list(2 * local_n_face + 1), local_n_face, face2_off_proc)

S5 face_off_proc = face1_off_proc + face2_off_proc

S6 n_off_proc = MAX(edge_off_proc, face_off_proc)

S7 sched_array(1) = face1_sched

S8 sched_array(2) = face2_sched

```

Figure 3.7: Inspector Code for Each Processor

Inspector Generation

Runtime support can be used either by a compiler or it can be embedded into distributed memory codes manually by programmers. The primitives carry out preprocessing that make it easy to produce parallelized loops that are virtually identical in form to the original sequential loops. Since the parallel and the sequential codes are virtually identical, it is possible to generate the same quality object code on the nodes of the distributed memory machine as produced by the sequential program running on a single node.

These primitives make use of hash tables [52] to recognize and exploit a number of situations in which a single off-processor distributed array reference is used several times. In such situations, the primitives fetch a single copy of each unique off-processor distributed

array reference.

The PARTI procedure *localize* carries out the bulk of the preprocessing needed to produce the executor code depicted in Figure 3.9. On each processor P, *localize* is passed:

1. A pointer to a paged distributed translation table (*translation_table* in S2),
2. A list of globally indexed distributed array references for which processor P will be responsible, (*part_edge_list* in S2), and
3. The number of globally indexed distributed array references ($2 * \text{local_n_edge}$ in S2).

Localize returns:

1. A schedule that can be used in PARTI gather and scatter procedures (*edge_sched* in S2),
2. An integer array that can be used to specify the pattern of indirection in the executor code (*local_edge_list* in S2), and
3. The number of distinct off-processor references found in *part_edge_list* (*edge_off_proc* in S2).

A sketch of how the procedure *localize* works is shown in Figure 3.8. The array *edge_list* shown in Figure 3.2 is partitioned between processors. The *part_edge_list* passed to *localize* on each processor in Figure 3.7 is a *subset of edge_list* depicted in Figure 3.2. *part_edge_list* cannot be used to index an array on a processor since *part_edge_list* refers to globally indexed elements of arrays *x* and *y*. *Localize* changes *part_edge_list* so that valid references are generated when the edge loop is executed. The buffer for each data array is placed immediately following the on-processor data for that array. For example, the buffer for data array *x* starts at $x(\text{n_on_proc}+1)$. Hence, when *localize* changes the *part_edge_list* to *local_edge_list*, the off-processor references are changed to point to the buffer addresses. When the off processor data are collected into the buffer using the schedule returned by *localize*, they are stored in such a way that execution of the edge loop using the *local_edge_list* accesses the correct data.

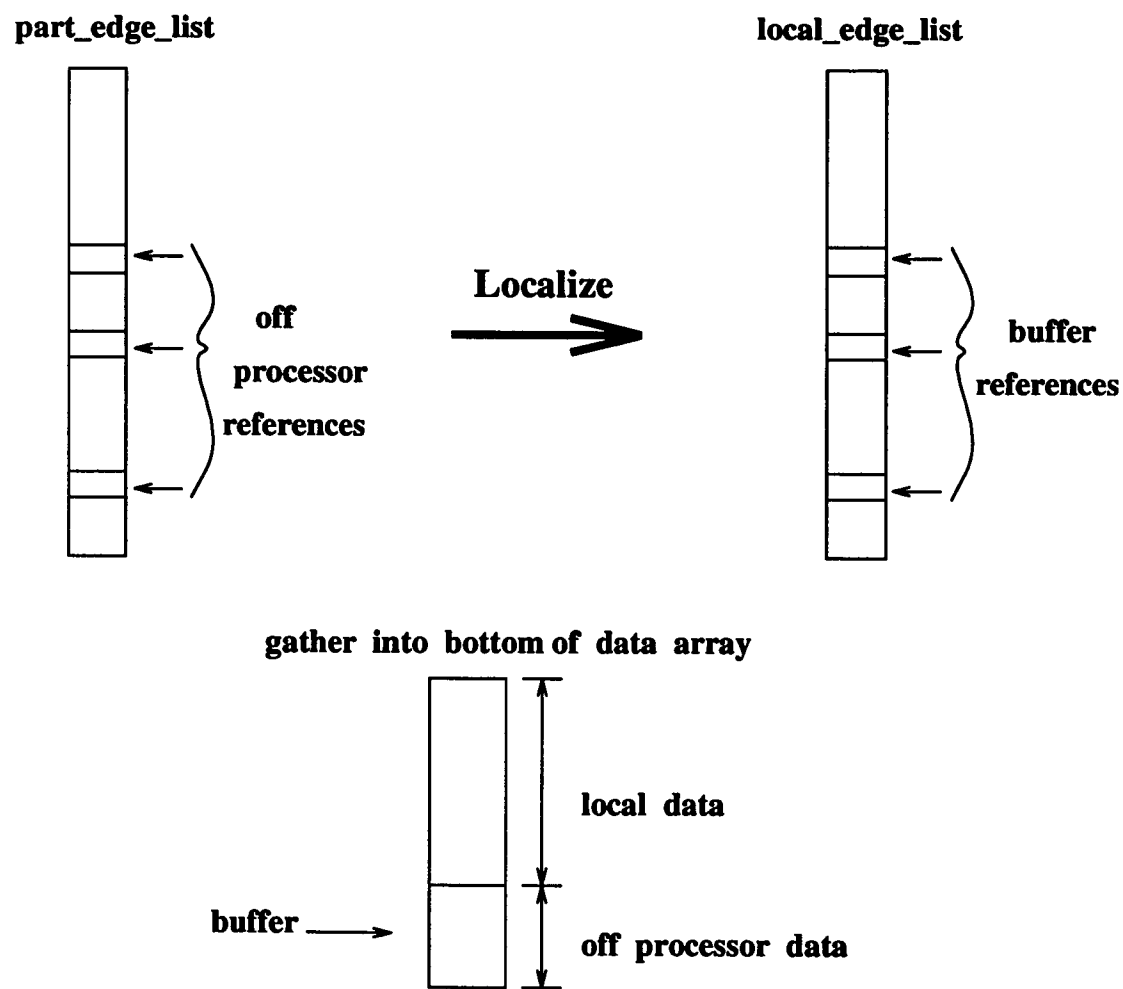


Figure 3.8: Localize Mechanism

A careful review of the face loop presented in Figure 3.2 shows that the distributed array **x** is indexed by **m1**, **m2** and **m3**, but the array **y** is indexed only by **m1** and **m2**. Two separate schedules are built, i.e., one with all **m1** and **m2** references and another with just **m3** references to be used to gather and scatter from the data arrays **x** and **y**. Hence, for the face loop there are two *localize* calls. Similar to the edge loop, the **face_list** in Figure 3.2 is partitioned between processors; each processor's share is represented by **part_face_list** in Figure 3.7 (statements S3, S4). The first call to *localize* (statement S3) generates a schedule for references **m1** and **m2**. The next call to *localize* (statement S4) builds a schedule for **m3** references.

In Figure 3.7, statement S5 is executed to find the total number of unique off-processor references made during the execution of the face loop. The largest number of unique off-processor references is stored in the variable **n_off_proc**. The **n_off_proc** value is required to obtain the total size of the **x** and **y** arrays that need to be allocated on each processor. Statements S7 and S8 are executed to store the face schedule into an array **sched_array** to be used later with the communication primitives.

Executor Generation

Figure 3.9 depicts the *executor* code with embedded Fortran callable PARTI procedures *dgather*, *dscatter_add* and *dmulti_gather*. Before the code is run, one must carry out the preprocessing phase described in Section 3.3.2. The executor code depends on the type of scheduling technique used. In the next section other types of scheduling techniques and their impact on the inspector and executor codes will be considered. The executor code shown in Figure 3.9 fetches unique off-processor values, considering one irregular loop at a time.

The arrays **x** and **y** are partitioned between processors; each processor is responsible for the long term storage of specific elements of each array. The way in which **x** and **y** are to be partitioned between processors is determined by the inspector. In the example, elements of **x** and **y** are partitioned between processors in exactly the same way. Each processor is

```

    real*8 x(n_on_proc + n_off_proc), y(n_on_proc + n_off_proc)
S1 call dgather(edge_sched, x(n_on_proc + 1), x)
S2 call clear_buffer(n_off_proc, y(n_on_proc + 1))
C Loop over edges involving x, y
L1 do i=1, local_n_edge
        n1 = local_edge_list(i)
        n2 = local_edge_list(local_n_edge + i)
        y(n1) = y(n1) + x(n1) + x(n2)
        y(n2) = y(n2) + x(n1) + x(n2)
    end do
S3 call dscatter_add(edge_sched, y(n_on_proc + 1), y)
C Loop over Boundary faces involving x, y
S4 call dmulti_gather(sched_array, 2, y(n_on_proc + 1), y)
S5 call clear_buffer(n_off_proc, y(n_on_proc + 1))
L2 do i=1, local_n_face
        m1 = local_face_list(i)
        m2 = local_face_list(local_n_face + i)
        m3 = local_face_list(2 * local_n_face + i)
        y(m1) = y(m1) + x(m1) + x(m2) + x(m3)
        y(m2) = y(m2) + x(m1) + x(m2) + x(m3)
    end do
S6 call dscatter_add(face1_sched, y(n_on_proc + 1), y)

```

Figure 3.9: Parallelized Code for Each Processor

responsible for *n_on_proc* elements of *x* and *y*.

It should be noted that, except for the procedure calls, the control structure of the loops in Figure 3.9 is identical to that of the loops in Figure 3.2. Though the names of the arrays *x* and *y* remain unchanged between the code shown in Figures 3.2 and 3.9, they represent different arrays. In Figure 3.2 the arrays *x* and *y* represent the global array. In Figure 3.9 the arrays *x* and *y* represent arrays local to the processor of a distributed memory. On each processor *P*, arrays *x* and *y* are declared to be larger than what would be needed to store the number of array elements for which *P* is responsible. Copies of off-processor array elements will be stored at the location beginning with local array elements *x*(*n_on_proc*+1) and *y*(*n_on_proc*+1). The extra elements are *overlap* regions [60] allocated to store off-processor elements.

The PARTI subroutine calls depicted in Figure 3.9 move data between processors using a precomputed communication pattern. The communication pattern is specified by either a single schedule or by an array of schedules. The procedure *dmulti_gather* takes an array of schedules as input and uses all of them to fetch off-processor data. The schedules specify the locations in distributed memory from which data are to be obtained. In Figure 3.9, off-processor data are obtained from array *x* defined on each processor. Copies of the off-processor data are placed in a buffer area beginning with *x*(*n_on_proc*+1).

The PARTI procedure *dscatter_add* in statements S3 and S6, Figure 3.9, accumulates data to off-processor memory locations. Both the *dscatter_add* calls obtain data to be accumulated to off-processor locations from a buffer area that begins with *y*(*n_on_proc*+1). Off-processor data are accumulated to locations of *y* between indices 1 and *n_on_proc*. When the accumulation for the face loop is done, using the *dscatter_add* function (statement S6), only the schedule *face1_sched* is used because it was the schedule set up using the references *m1* and *m2*. In Figure 3.9 statements S2 and S5 are calls to a function *clear_buffer*. The calls are made to initialize the buffer location of the array *y* to 0.0.

3.3.3 Incremental Scheduling

In most scientific applications, the computational domain is discretized and physical quantities, like velocity, pressure, charge etc., are evaluated at discrete points in the domain over a period of time. Usually the variables, which represent the quantities of interest are evaluated once at the end of each timestep. In the following timestep, these variables are used to calculate their new values. Hence there are situations where in a single timestep, multiple loops access the same data.

A scheduling technique called incremental scheduling has been developed allowing access to only those off-processor data that do not already exist in the processor. In this section, the preprocessing required to generate an incremental schedule is described. The preprocessing required to parallelize the code shown in Figure 3.2 using incremental scheduling is depicted in Figure 3.11 and the executor code is shown in Figure 3.12.

Incremental Inspector

In Figure 3.2 no assignments to x are carried out. In the beginning of the execution of both the loops L1 and L2, each processor can gather a single copy of every distinct off-processor value of x referenced by these loops. The PARTI procedure *multi.localize* (S4 in Figure 3.11) makes removing these duplicate references simple. The procedure *multi.localize* makes it possible to obtain only those off-processor data not requested by a given set of pre-existing schedules. The returned schedules can be utilized by the communication routines to bring in the required data.

A pictorial representation of the *incremental schedule* is given in Figure 3.10. The schedule to bring in the off-processor data for the *edge_loop* is given by the *edge schedule* and is formed first. During the formation of the schedule to bring in the off-processor data for the *face_loop* the duplicates are removed, shown by the shaded region in Figure 3.10. Removal of duplicates is achieved by using a hash table. The off-processor data to be accessed by the *edge schedule* are first hashed using a simple hash function. Next, the data to be accessed during the *face_loop* are hashed. At this point, the information that exists

**OFF PROCESSOR FETCHES
IN SWEEP OVER EDGES**

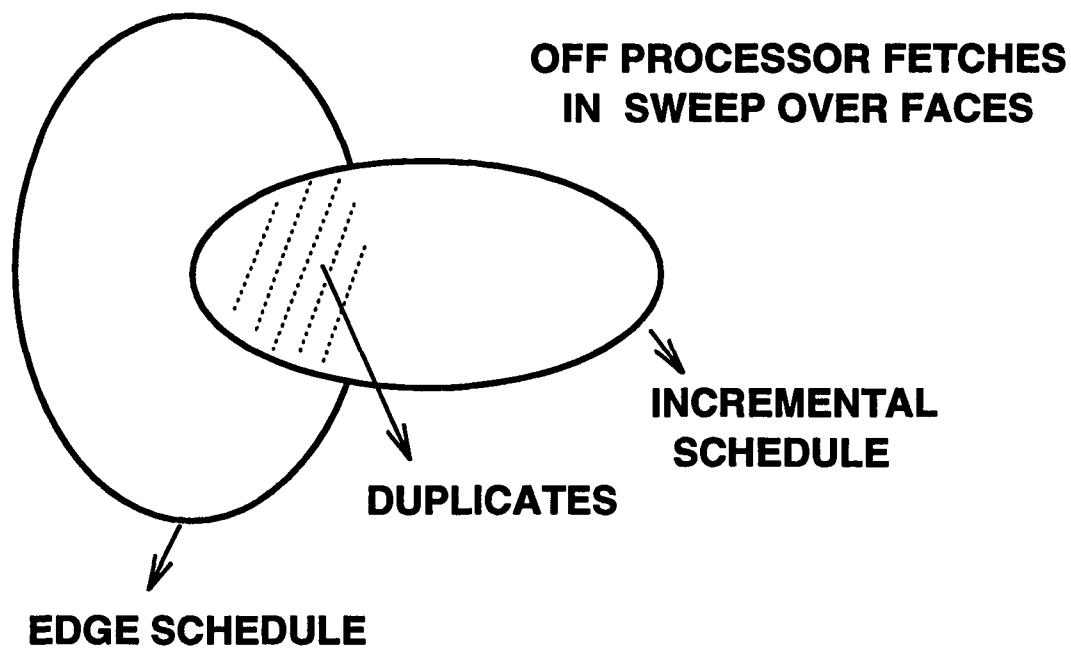


Figure 3.10: Incremental Schedule

```

S1 translation_table = BuildDsTable(myvals, on_proc, replication_factor)

S2 call localize(translation_table, edge_sched, part_edge_list, local_edge_list,
                2 * local_n_edge, edge_off_proc)

S3 sched_array(1) = edge_sched

S4 call multi_localize(translation_table, face_sched, incremental_face_sched,
                part_face_list, local_face_list, 3 * local_n_face, face_off_proc,
                new_face_off_proc, buffer_mapping, 1, sched_array)

S5 sched_array(2) = incremental_face_sched

S6 n_off_proc = MAX(edge_off_proc , face_off_proc)

```

Figure 3.11: Inspector Code for Each Processor Using Incremental Scheduling

in the hash table allows removal of all the duplicates and formation of the *incremental schedule*. In the Section 4.4 results showing the usefulness of incremental schedule will be presented.

The inspector code is shown in Figure 3.11. The first call after the translation table has been generated is made to the function *localize* to generate the schedule for the edge loop (*edge_sched*). During formation of the incremental schedule for the face loop, the information in the schedule for the edge loop is utilized. To review the work carried out by *multi_localize*, the significance of all but one of the arguments of this PARTI procedure will be summarized. On each processor *multi_localize* is passed:

1. A pointer to a paged distributed translation table (*translation_table* in S4),
2. A list of globally indexed distributed array references (*face_list* in S4),
3. The number of globally indexed distributed array references ($3 * \text{local_n_face}$ in S4),

4. The number of pre-existing schedules that need to be taken into account when removing duplicates (1 in S4), and
5. An array of pointers to pre-existing schedules (`sched_array` in S4).

Multi_localize returns:

1. A *schedule* that can be used in PARTI gather and scatter procedures. This schedule *does not take any pre-existing schedules into account* (`face_sched` in S4),
2. An *incremental schedule* that includes only off-processor data accesses not included in the pre-existing schedules (`incremental_face_sched` in S4),
3. A list of integers that can be used to specify the pattern of indirection in the executor code (`local_face_list` in S4),
4. The number of distinct off-processor references in `face_list` (`face_off_proc` in S4), and
5. The number of distinct off-processor references not encountered in any other schedule (`new_face_off_proc` in S4).

Incremental Executor

The procedure *dmulti_gather* in the executor in Figure 3.12 obtains off-processor data using *two schedules*; *edge_sched* produced by *localize* (S2 Figure 3.11) and *incremental_face_sched* produced by *multi_localize* (S4 Figure 3.11). The procedure *dmulti_gather* has already been discussed in Section 3.3.2 but nothing has been said so far about the distinction between *dscatter_add* and *dscatter_addnc*. When making use of incremental schedules, a single buffer location is assigned to each off-processor distributed array element. For the example, separate off-processor accumulations are carried out after loops L1 and L2. As described below, the off-processor accumulation procedures may no longer reference consecutive elements of a buffer.

Copies of distinct off-processor elements of *y* are assigned to buffer locations, to handle off-processor writes in loop L1, Figure 3.12. Then a schedule (*edge_sched*) can be used

```

    real*8 x(n_on_proc + n_off_proc), y(n_on_proc + n_off_proc)
S1 call dmulti_gather(sched_array, 2, x(n_on_proc + 1), x)
S2 call clear_buffer(n_off_proc, y(n_on_proc + 1))

    C Loop over edges involving x, y
L1 do i=1, local_n_edge
        n1 = local_edge_list(i)
        n2 = local_edge_list(local_n_edge + i)
        y(n1) = y(n1) + x(n1) + x(n2)
        y(n2) = y(n2) + x(n1) + x(n2)
    end do

S3 call dscatter_add(edge_sched, y(n_on_proc + 1), y)

    C Loop over Boundary faces involving x, y
S4 call clear_buffer(n_off_proc, y(n_on_proc + 1))
L2 do i=1, local_n_face
        m1 = local_face_list(i)
        m2 = local_face_list(local_n_face + i)
        m3 = local_face_list(2 * local_n_face + i )
        y(m1) = y(m1) + x(m1) + x(m2) + x(m3)
        y(m2) = y(m2) + x(m1) + x(m2) + x(m3)
    end do

S5 call dscatter_addnc(face_sched, y(n_on_proc + 1), buffer_mapping, y)

```

Figure 3.12: Parallelized Code for Each Processor Using Incremental Scheduling

to specify where in distributed memory each consecutive value in the buffer is to be accumulated. PARTI procedure *dscatter_add* can be employed; the procedure uses schedule *edge_sched* to accumulate to off-processor locations consecutive buffer locations beginning with $y(n_on_proc + 1)$. When off-processor elements of y are assigned to buffer locations in L2, some of the off-processor copies may already be associated with buffer locations (done in loop L1). Consequently, in S3, Figure 3.12, the schedule (*face_sched*) must access buffer locations in an irregular manner. The pattern of buffer locations accessed is specified by integer array *buffer_mapping* passed to *dscatter_addnc* in S3, Figure 3.12 (*dscatter_addnc* stands for *dscatter_add* non-contiguous).

Chapter 4

Performance Analysis of Runtime Support

A set of procedures has been produced that support a type of weakly coherent distributed shared memory; these procedures can be coupled closely to distributed memory compilers. These primitives (1) coordinate interprocessor data movement, (2) manage the storage of and access to copies of off-processor data (3) minimize interprocessor communication requirements and (4) support a shared name space. In this chapter a detailed performance and scalability analysis of the communication primitives are discussed. This chapter also presents performance data obtained from parallel implementation of adaptive and non-adaptive irregular applications.

This chapter describes and systematically evaluates all the optimizations that have been incorporated into the tools. The optimizations reduce communication latency and volume. Performance data for the paged distributed translation table described in Section 3.3.1, are also presented.

Performance of optimizations are characterized by using

- Synthetic workloads,
- Test loops with data access patterns drawn from unstructured applications, and

- Real applications.

The synthetic workload was developed jointly with a group from University of Maryland [37]. Synthetic workloads can be used to characterize the performance of the optimizations under a wide variety of conditions.

In sparse and unstructured computations, distributed arrays are typically accessed using indirection. In many cases (e.g. distributed arrays referenced in loops with no loop carried dependencies or distributed arrays referenced in loops with accumulation type dependencies), it is possible to prefetch required off-processor data before a loop is executed. Sometimes several loops access the same off-processor memory locations. As long as it is known that the values assigned to off-processor memory locations remain unmodified, it is possible to reuse stored off-processor data. A mixture of compile-time and run-time analysis can be used to generate efficient code for irregular problems [36, 101]. This chapter provides a detailed description of communication optimizations that prove to be useful for optimizing irregular problem performance. The PARTI primitives described in the previous chapter incorporate all the communication optimizations that will be presented in this chapter.

The class of problems considered in this thesis consists of a sequence of clearly demarcated concurrent computational phases, where data access patterns cannot be anticipated until runtime, and these problems are called static irregular concurrent computations [14]. In these problems, once runtime information is available, 1) data access patterns are known before each computational phase and 2) the same data access patterns occur many times. Adaptive problems can fall into this class of problems as long as data access patterns change relatively infrequently. A typical loop in such computations is shown in Figure 4.1. In this loop, the arrays *x*, *y*, *ia* and *ib* are all distributed arrays. The arrays *ia* and *ib* are used to index the arrays *x* and *y*, respectively. At compile time, it is not possible to determine the indices of *x* and *y* that are accessed because they are dependent on the values stored in the arrays *ia* and *ib*. The data access pattern becomes available at runtime. Runtime compilation techniques are used to parallelize such loops.

```
DO i = 1, n  
  
    x(ia(i)) = x(ia(i)) + y(ib(i))  
  
end do
```

Figure 4.1: Simple Irregular Loop

4.1 Applications Used for Performance Analysis

In this section the applications that are utilized in the performance studies are briefly described. Both real applications and artificial workloads are used to learn the behavior of the tools in various situations.

4.1.1 Real Applications

Many scientific codes have been implemented on parallel machines, using the PARTI primitives. In this section, two application codes are briefly described, stating how they stress the primitives. In Section 4.1.1, an explicit Euler solver [75, 33] developed at ICASE by Dimitri Mavriplis is described. Section 4.1.1 describes the molecular dynamics code CHARMM [19, 34]. Both these codes have been implemented on the Intel Gamma and Delta machines.

Unstructured Euler Kernel

Unstructured meshes provide a great deal of flexibility in discretizing complex domains and offer the possibility of easily performing adaptive meshing. However, unstructured mesh problems result in large sparse matrices and if the problems are to be executed on a distributed memory machine, one would require runtime preprocessing. The connectivity of the meshes is quite low, when compared with the connectivity that is generated for other problems, such as molecular dynamics or particle dynamics.

The unstructured Euler code solves the three dimensional compressible gas dynam-

ics equations. The solution technique has been outlined in [75, 58]. The equations are discretized on an unstructured mesh using a Galerkin finite-element technique. The flow variables are stored at the vertices of the mesh. However, certain precautions have to be taken in order to stabilize the solution. The spatially discretized equations are integrated to a steady state, using a 5-step Runge-Kutta timestepping method. The program comprises loops over the edges and faces of the three dimensional unstructured mesh. A multigrid solution technique can be used to speedup the solution time.

Molecular Dynamics

Molecular dynamics (MD) is a technique for simulating the thermodynamic and dynamic properties of liquid and solid systems. For each timestep of the simulation, two separate calculations are performed. The first part deals with the bonded and non-bonded force calculations for each atom. The second part is the integration of the Newton equation for each atom. In most MD codes, the bulk of the time (a little more than 90%) is spent in the long-range force, i.e., the non-bonded force calculation. Hence the non-bonded force calculation needs to be parallelized efficiently. The non-bonded force calculation uses an $O(N^2)$ algorithm, where N represents the number of atoms. Every single atom interacts with each other, but usually a cutoff distance R_c is specified and interactions outside the cutoff are neglected. The non-bonded force calculation has two distinct parts. For each atom, first the pairlist (atoms within R_c distance) is generated; next, the Vander Waals and electrostatic force calculations are performed. The pairlist generation is not performed every iteration but after every n iteration, where n is a variable that can be fixed by the user.

The MD code used in this case was CHARMM (Chemistry at HARvard Macromolecular Mechanics) [19], and it was developed at Harvard University for biomolecular simulations. The program is relatively efficient, and it uses empirical energy functions to model molecular systems. Written in Fortran, the code is about 110,000 lines long and is capable of performing a wide range of analyses. The important simulation routines are the dynamic

analysis, the trajectory manipulations, energy calculations and minimization, and vibrational analysis. The program also performs statistical analysis, time series and correlation function analysis as well as spectral analysis.

4.1.2 Synthetic Workload

A synthetic workload was developed as part of a group project done at the University of Maryland [37]. A parameterized workload generator was developed to simulate the kinds of data reference patterns and communication characteristics encountered in concurrent irregular scientific problems. The synthetic workload consists of two parts, the Communication Pattern Generator (CPG) and the Data Access Pattern Generator (DAPG). The CPG is used to define the communication pattern induced by the problem. The DAPG generates indirection arrays that embody the communication pattern specified by the CPG.

A communication graph $G = (V, E, w)$ is a weighted graph where vertices correspond to individual distributed partitions. For any two partitions $u, v \in V$ there exists an edge $(u, v) \in E$ iff the partitions u and v need to communicate with each other at runtime. The volume of the communication is determined by the weight function $w : E \rightarrow \mathcal{N}$. The generation of the communication graph is controlled by the following parameters:

- Connectivity (C)

Connectivity is the average degree of vertices in G . $C = (\sum_{v \in V} \deg(v)) / |V|$, where $\deg(v)$ is the degree of vertex v . Connection between two partitions causes communication to occur at runtime between the partitions. The connectivity parameter is translated into the total number of distinct messages for each processor to send or receive at each phase of the computation.

- Total Volume of Communication (V).

Total volume of communication limits the assignment of weights to the edges in a communication graph, so that the sum of the weight of all the edges in a communication graph is equal to V . In the generation of the communication graphs for the

experiments, V is used to distribute the weights to edges in a uniform way. That is,
 $\forall e \in E : w(e) = V/|E|$.

- Manhattan Distance (D).

The Manhattan Distance, D , specifies that no two partitions more than D apart can be connected via an edge in the communication graph. The partitions in a real world problem are physically related to each other often with a relationship that is determined by the specific problem. These problem-imposed relationships can be represented as a graph, called a problem topology graph, in which vertices represent partitions and edges represent the relationships between partitions. It is usually the case that, in the problem domain, partitions are scattered in 2 or 3 dimensional space. The problem topology graphs, often representing the physical proximity of partitions, arise from the fact that distant partitions have little or no relationships with each other. The Manhattan Distance of two partitions is defined as the sum of the canonical distances between them in the problem topology graph. For example, if the relationships between partitions are represented by a 2D-grid, the Manhattan Distance of u and v is $|u_x - v_x| + |u_y - v_y|$.

It should be noted, however, that more sophisticated communication models can be defined to replicate the communication behavior of irregular problems. One extension is the addition of extra features such as variability of the connectivity and communication volume. However, the current model is general enough to illustrate the key performance parameters of the optimization primitives.

The second part of the synthetic workload generator is the Data Access Pattern Generator (DAPG), which is responsible for generating the data access patterns utilizing the communication graph. The actual communication takes place in a way determined by the communication graph. The data access pattern is defined to be a permutation of a subset of the global index space. It specifies which global data indices have been accessed locally. The output of the DAPG is a set of indirection arrays that will be used in accessing the

distributed arrays, whereas the input is the communication graph generated by the CPG and the following parameters :

- Number of Loops (N_{loop})

The number of loops determines the number of consecutive test loops associated with the DAPG. The test loops are of the form depicted in Figure 4.2, where two consecutive computational phases exist, so N_{loop} is equal to 2. For each computational phase, the DAPG produces N_{loop} indirection arrays to access the loop's distributed arrays.

- Intersection Ratio (R_{int})

The intersection ratio of two indirection arrays is defined as the ratio of the number of identical global data indices the indirection arrays contain over their size. Its result is the degree to which data usage patterns in two indirection arrays are similar. If R_{int} is zero, the global data indices stored in two different indirection arrays are completely disjoint, and two indirection arrays are exactly the same if R_{int} is 1.0.

- Number of Duplicates (N_{dup})

The number of duplicates for a given indirection array is defined to be the number of distinct occurrences of the same off-processor data reference. If the number of duplicates for an indirection array is 2, each unique reference in the indirection array will occur twice. Note that this parameter has no effect on the total volume of unique data communicated.

- Number of Dimensions (N_{dim})

N_{dim} measures the degree of reuse of the same data access pattern across the dimensions of a distributed array.

A summary of the symbols that are part of the workload generator and their meaning are presented in the Table 4.1.

An example of the type of workload generated is shown in Figure 4.2. For this case, the various inputs to the DAPG are shown in Table 4.2. Since R_{int} equals 0.5, half the

Table 4.1: Summary of Symbols used in the Workload.

Symbol	Meaning
C	Connectivity
V	Total Volume of Communication
D	Manhattan distance between partitions
R_{int}	Intersection Ratio
N_{dup}	Duplication Factor
N_{loop}	Number of test loops
N_{dim}	Number of identically referenced distributed array slices in each test loop
P	Total Number of Processors

First Loop

do i = 1, n

$x(\mathbf{ia}(i), 1) = x(\mathbf{ia}(i), 1) + z(\mathbf{ia}(i), 1)$

$x(\mathbf{ia}(i), 2) = x(\mathbf{ia}(i), 2) + z(\mathbf{ia}(i), 2)$

end do

Second Loop

do i = 1, n

$x(\mathbf{ib}(i), 1) = x(\mathbf{ib}(i), 1) + z(\mathbf{ib}(i), 1)$

$x(\mathbf{ib}(i), 2) = x(\mathbf{ib}(i), 2) + z(\mathbf{ib}(i), 2)$

end do

Figure 4.2: Synthetic Workload Loops

Table 4.2: DAPG parameters for Synthetic Workload Loops.

Symbol	Value
R_{int}	0.5
N_{dup}	2
N_{loop}	2
N_{dim}	2

values stored in array **ia** are also present in the array **ib**. Since N_{loop} equals 2, there are two indirection arrays namely, **ia** and **ib**. N_{dim} equals 2, making the the upper bound of the compressed dimension of all the data arrays 2 (in this case $x(*,2)$ and $z(*,2)$). Since N_{dup} equals 2, each reference in **ia** is repeated twice. The same follows for **ib**.

4.2 Communication Optimizations

In this section, communication optimizations developed for this thesis are presented. Section 4.2.1 shows how *software caching* can be used to reduce the volume of communication between processors. One such optimization is to remove redundant off-processor accesses associated with a particular indirect array reference. A more aggressive optimization removes redundant off-processor accesses associated with several indirect array references. Section 4.2.2 describes the optimizations developed to reduce communication startups by coalescing communications into a decreased number of messages.

4.2.1 Software Caching

During the execution of irregular loops on distributed memory (or distributed shared memory) machines, the same off-processor data may be accessed repeatedly. In many cases, data needed by an array reference can be prefetched before a loop's computation begins. In other cases, data needed by a set of irregular references to the same array can be prefetched. In either case, the same off-processor data may be accessed multiple times, but only a single copy of the data need to be fetched from off-processor. The process of prefetching off-processor data and storing it locally is software caching. Informally, the prefetches can be

carried out when

- it is possible to predict array reference patterns prior to a loop's execution, and
- it is known that all array data subject to prefetch remains live, i.e., there is no possibility that the prefetched values are no longer valid.

There are two ways of managing software caching, "simple" and "incremental."

Simple Software Caching

A hash table is utilized to identify duplicate off-processor data accesses associated with the indirect references to a single data element. A simple hash function (**mod** operator) is used. Communication schedules are generated from the lists of unique off-processor data accesses. These schedules store the communication patterns to be used by the gather and scatter primitives. During the schedule generation process, each processor sends the lists of data it needs from all other processors; it also receives the lists of data it must send to other processors. These lists contain the indices of the data that need to be communicated. Each schedule is associated with a distribution and a data access pattern, rather than being tied to specific data arrays. Hence, if there exists two references to different arrays, where the arrays are distributed in the same way and the data access patterns are identical, the same schedule can be used to gather or scatter data to these arrays.

Incremental Scheduling

Data communication volume is reduced by tracking and reusing live off-processor data copies. In a number of application codes, multiple indirect references occur to the same data array. When it is known that no array assignments can occur between some set of indirect references, i.e., the array in question remains live between the indirect references, then, only a single copy of each unique off-processor value needs to be fetched.

Assume there are N different indirect array references to any distributed data array D . From each reference, off-processor indices used to access data from the array D can

be obtained. Let IA_I be the set of off-processor indices from reference I . Hence, $IA = \{IA_1, IA_2, \dots, IA_N\}$ is the set of the sets of off-processor indices used to access data from D . The use of incremental schedules allows one to bring in only the data that are not available locally:

$$\bigcup IA_I = \{ ia : ia \in IA_I \text{ for some set } IA_I \in IA \}.$$

The number of indices belonging to the set, IA_I , is potentially smaller than the number of indices one would get by simply concatenating the indices obtained from separately applying simple software caching to each distributed array reference. If every index listed in each of the set IA is different, then there is no advantage in doing incremental scheduling. On the other hand, if there is significant overlap in the off-processor references obtained from the reference sets, then a large reduction in communication volume is achieved by using incremental scheduling.

4.2.2 Communication Coalescing

One can frequently collect many data items destined for the same processor into a single message. This kind of optimization is sometimes called *communication coalescing*. The object of communication coalescing is to reduce the number of message startups. For many distributed memory systems, there is a substantial latency associated with message passing. For instance, Bokhari [15] measured the time to communicate a message of size k (bytes) between two nodes of an Intel iPSC/860, as

$$T = 65.0 + 0.425k + 10.0h, \text{ for } 0 < k \leq 100, \text{ and}$$

$$T = 147.0 + 0.390k + 30.5h, \text{ for } k > 100$$

where T is the time in μsecs and h is the number of hops between the communicating processors. On the Intel iPSC/860, the cost of a startup latency is equal to the cost of sending one to several hundred bytes. The three types of communication coalescing are

- Simple Communication Aggregation,

- Communication Vectorization, and
- Schedule Merging.

Simple Communication Aggregation

It is frequently possible to anticipate which data must be communicated before a loop executes. Preprocessing is needed to characterize the data required by a given right-hand side array reference. Prior to a loop's execution, all the data that each pair of processors need to exchange is packed into a single message. In a similar manner, the communication (and accumulations) associated with left hand side array references can often be deferred until after a loop's computation. This optimization may be referred to as *simple communication aggregation*.

Communication Vectorization

If a number of columns of a multi-dimensional array are distributed in a conforming manner, and if the data access patterns from these columns are the same, then the primitives gather and scatter data from all the columns using a single communication phase. The optimization does not reduce the communication volume but reduces the startup latency. Hence, if any processor P is to receive data from N processors for L columns then the reduction of startup latency time is given by

- $\text{Latency_Reduction} = N * \text{Time}_{\text{latency}} * (L - 1).$

The PARTI primitives for multi-dimensional arrays perform communication vectorization.

Schedule Merging

When data are gathered from or scattered to the same data array using a number of different schedules, then the schedules can be merged to reduce the number of message startups and thereby the latency. Schedule merging is orthogonal to the software caching optimizations; for instance, one can merge sets of schedules that arise from simple software caching or sets

of schedules obtained from incremental scheduling. The total reduction in latency is by the factor $(S - 1)$, where the number of merged schedules is S . PARTI provides primitives that merge a number of schedules to form a single communication schedule.

4.2.3 Example Test Codes

The application of the runtime support depends on the nature of the communication optimization. The type of communication optimization to be used at any particular situation has to be determined by the compiler. Depending on compile time analysis, calls to the correct runtime support routines have to be made by the parallelizing compiler. For instance, in Figures 4.3 and 4.4 the test loops associated with simple communication aggregation are compared to schedule merging. The simple communication aggregation case shown in Figure 4.3 does the preprocessing with the various indirection arrays at the beginning. It returns four schedules, one for each of the indirection arrays. The z values are fetched immediately before each loop executes; the schedule for ic is employed before the first loop, and the schedule for id is employed before the second loop. After the execution of the first loop, the off-processor x values are accumulated using the schedule for ia . Similarly, after the second loop's computation, the off-processor accumulation of x values are done by using the schedule for ib .

The schedule merging code is shown in Figure 4.4. As in the previous case schedules are built using all the indirection arrays. In this case, the schedules are merged, and instead of four, there are two schedules, one for ia and ib and one for ic and id . All the required values of z are fetched using vectorized communication (z being a multi-dimensional array) before execution of the loops. Off-processor values of x are accumulated by using the schedule for ia and ib after both loops execute. Accumulation can be delayed until the completion of execution of both the loops because of the commutative property of the '+' operator. The executor communication cost, when schedule merging and vectorization are performed, is much lower than that of the simple software caching. The inspector cost for schedule merging is higher than the inspector cost of the software caching.

Preprocessing for indirection arrays ia, ib, ic and id

gather the values of array z using schedule for ic

do i = 1, n

$x(ia(i)) = x(ia(i)) + z(ic(i))$

end do

accumulate values of x using schedule for ia

gather the values of array z using schedule for id

do i = 1, n

$x(ib(i)) = x(ib(i)) + z(id(i))$

end do

accumulate values of x using schedule for ib

Figure 4.3: Simple Communication Aggregation Case

Note that while the software caching and communication coalescing optimizations are orthogonal, on distributed memory machines it makes sense to use certain optimizations together. For instance, if incremental scheduling is employed, one can easily produce a single merged schedule to perform the communication of the unique off-processor elements, identified by the incremental scheduling process.

Preprocessing to build a single schedule using arrays ia and ib

Preprocessing to build a single schedule using arrays ic and id

Gather for z using the single schedule for arrays ic and id

do i = 1, n

$x(\mathbf{ia}(i), 1) = x(\mathbf{ia}(i), 1) + z(\mathbf{ic}(i), 1)$

$x(\mathbf{ia}(i), 2) = x(\mathbf{ia}(i), 2) + z(\mathbf{ic}(i), 2)$

end do

do i = 1, n

$x(\mathbf{ib}(i), 1) = x(\mathbf{ib}(i), 1) + z(\mathbf{id}(i), 1)$

$x(\mathbf{ib}(i), 2) = x(\mathbf{ib}(i), 2) + z(\mathbf{id}(i), 2)$

end do

Accumulate x using the single schedule for arrays ia and ib

Figure 4.4: Schedule Merging Case

4.3 Scaling Characteristics of the Optimizations

Optimizations are applied to data access patterns generated when a given unstructured problem is mapped onto a multiprocessor. Measured performance on a given architecture consequently depends on

- the nature of the unstructured code (e.g., the real codes outlined in Section 4.1.1 or the test loops in Figure 4.2),
- the dataset (e.g., the data structures used to represent unstructured meshes and molecular interactions described in Section 4.1.1), and
- the way in which the dataset is partitioned among processors.

In this section, effects of the various optimizations on unstructured problem communication requirements are examined. In the analysis presented in this section, the synthetic workload described in Section 4.1.2 is used, which employs a set of loops of the type depicted in Figure 4.2. In the experimental analysis presented in the following sections, both the synthetic workload and data access patterns derived from real applications are utilized.

The volume of communication and the number of communication startups associated with bringing in off-processor data are presented in Table 4.3. The row labeled “naive” stands for no optimization at all; each processor requests its data whenever that data is needed locally. In the “naive” case, the number of communication startups is equal to the number of data elements communicated. From Section 4.1.2, recall that V/P represents the volume of communication that must be sent and received by each processor, N_{loop} represents the number of test loops employed by the Data Access Pattern Generator, and N_{dim} represents the number of identically referenced array slices. When targeted at distributed memory architectures, the naive implementation is extremely inefficient (see [94]).

The row labeled “simple communication aggregation” gives the communication characteristics associated with the optimization described in Section 4.2.2. The optimization reduces the number of messages that must be transmitted. For each array slice (N_{dim}) and

each test loop (N_{loop}), every processor must communicate with each of the neighboring C processors. Note that the optimization does nothing to reduce communication volume. The optimization reduces latency costs compared to the naive implementation, but incurs two other costs: The costs are the memory overhead of storing the schedules associated with the communication and the pre-processing overhead for precomputing the communication requirements in the irregular computation.

The next optimization depicted in Table 4.3, labeled “simple software caching,” includes both simple software caching (Section 4.2.1) along with simple communication aggregation. Simple software caching involves eliminating intra-loop duplicates. The addition of this optimization reduces the communication time and space requirements compared to the simple communication aggregation case. The trade-off is the extra preprocessing required by the inspector and the memory required for the hash table. The communication volume for simple software caching is a factor of N_{dup} smaller than the communication volume for simple communication aggregation.

The next optimization depicted in Table 4.3, labeled “communication vectorization,” includes communication vectorization (Section 4.2.2) along with simple software caching and simple communication aggregation. The addition of the communication vectorization optimization leaves the communication volume unchanged but reduces the number of startups by an additional factor of N_{dim} . The next row of the table, “schedule merging,” adds the schedule merging optimization (Section 4.2.2) to the optimizations represented in the rows above. The “schedule merging” optimization makes it possible to prefetch all data needed by the entire set of test loops before executing the first of the test loops. The number of startups in this case is reduced by a factor of N_{loop} and is equal to C .

Finally, the incremental scheduling optimization (Section 4.2.1) is added to the optimizations mentioned above. Incremental scheduling allows one to fetch from off-processor only the unique data values needed by any one of the test loops and it produces a savings when more than one test loop uses the same datum.

Table 4.3: Executor Communication Requirements (Gather, Scatter or Accumulate)

<i>Optimization</i>	<i>Communication</i>	
	Volume	Number Startups
Naive	$\frac{N_{dim}N_{loop}V}{P}$	$\frac{N_{dim}N_{loop}V}{P}$
Simple Communication Aggregation	$\frac{N_{dim}N_{loop}V}{P}$	$N_{dim}N_{loop}C$
Simple Software Caching	$\frac{N_{dim}N_{loop}V}{PN_{dup}}$	$N_{dim}N_{loop}C$
Communication Vectorization	$\frac{N_{dim}N_{loop}V}{PN_{dup}}$	$N_{loop}C$
Schedule Merging	$\frac{N_{dim}N_{loop}V}{PN_{dup}}$	C
Incremental Scheduling	$\frac{N_{dim}V}{PN_{dup}(2-R_{int})^{N_{loop}-1}}$	C

The left hand side array references in the test loops in Figure 4.2, involve accumulations. In most cases, experience with real applications has indicated that it is permissible to defer off-processor accumulations until after a loop. The deferring of accumulations until after the loop has the effect of changing the order in which the accumulations are carried out. In the author's experience, the change in operation order does not usually cause problems, since such loops are routinely vectorized, and vectorization also changes the order in which values are accumulated. Limited to carrying out deferred accumulations after each loop, it is found that the schedule merging and incremental scheduling optimizations cannot be employed. In some applications, such as molecular dynamics, programmers find that they can defer accumulations until after a sequence of (non-dependent) loops are executed. In these cases, one could make use of schedule merging and incremental scheduling optimizations.

The communication requirements associated with preprocessing are very closely tied to the communication requirements needed to execute off-processor gathers, scatters and accumulations. Table 4.4 depicts these communication requirements. Some advantage is gained from the fact that the same schedule can be reused each time communication is

Table 4.4: Inspector Communication Requirements (Gather, Scatter or Accumulate)

Optimization	Communication	
	Volume	Number Startups
Naive	-	-
Simple Communication Aggregation	$\frac{N_{loop}V}{P+N_{loop}C}$	$2N_{loop}C$
Simple Software Caching	$\frac{N_{loop}V}{PN_{dup}+N_{loop}C}$	$2N_{loop}C$
Communication Vectorization	$\frac{N_{loop}V}{PN_{dup}+N_{loop}C}$	$2N_{loop}C$
Schedule Merging	$\frac{N_{loop}V}{PN_{dup}+N_{loop}C}$	$2N_{loop}C$
Incremental Scheduling	$\frac{V}{PN_{dup}(2-R_{int})^{N_{loop}-1}+C}$	$2C$

carried out for identically referenced, identically distributed arrays (or array sections). In the case of the test loops, it is clear that the preprocessing for identically distributed array sections need not be repeated. The advantage is reflected in the communication volume and startup numbers depicted in Table 4.4.

4.4 Experiments and Results

This section describes the experiments performed and the corresponding results. A number of different experiments were performed using the synthetic workload generator and the application code kernels. The results show the performance of the primitives and also how they scale with the increase in the number of processors. All experiments were executed on the Intel Gamma machine, and the number of processors ranged from 32 to 128.

4.4.1 Synthetic Workload Performance Results

Empirical performance results to characterize the effectiveness of the communications optimizations are presented in Section 4.2.

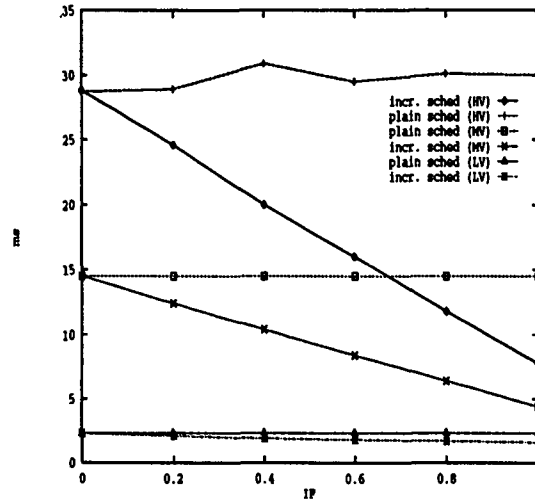


Figure 4.5: Cost vs. volume of communication and Intersection Ratio (gather, $\mathcal{N}_{loop}=4$, $P = 32$, $\mathcal{R}_{int} = IF$)

Comparison of Communication Optimizations

The reduction in communication time associated with incremental schedules is shown in Figure 4.5. Performance of a code which employs schedule merging with incremental scheduling,

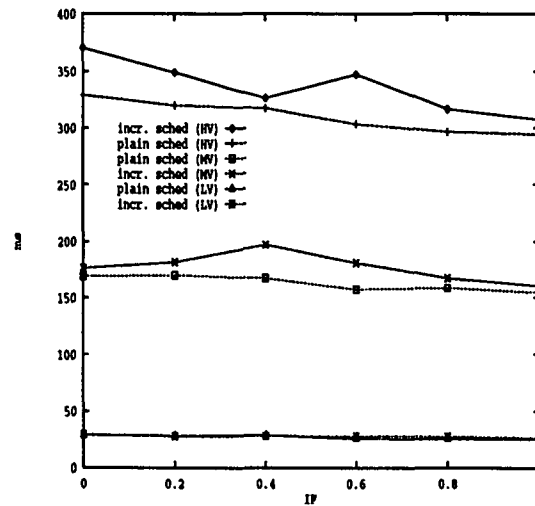


Figure 4.6: Cost vs. volume of communication and Intersection ratio (Inspector, $\mathcal{N}_{loop}=4$, $P = 32$, $\mathcal{R}_{int} = IF$)

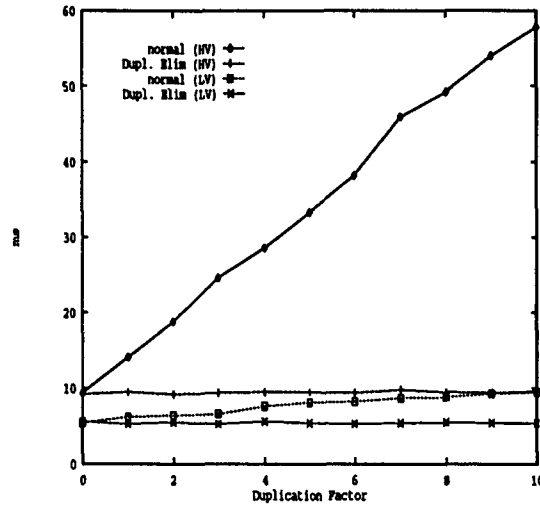


Figure 4.7: Cost vs Number of duplicates (Executor, low \mathcal{N}_{dup} , 32 Processors)

versus simple software caching carried out separately for each loop presents an interesting comparison. Four loops are used in the test loop code ($N_{loop} = 4$). The communication graph is kept constant ($C=4$) but the \mathcal{R}_{int} parameter is varied in order to change the number of shared off-processor accesses. The loop structure is similar to the one presented in Figure 4.4. The experiment is repeated for low (≈ 100 floating point numbers), medium (≈ 1000 floating point numbers), and high (≈ 2000 floating point numbers) communication volume. The results shown in the different graphs are obtained from experiments executed on a 32 processor Intel Gamma machine. Figure 4.5 gives the timings for the *gather* calls both for incremental scheduling and simple software caching. For both high and medium communication volumes, communication time for the incremental case drops rapidly as the intersection ratio becomes close to 1. The inspector times are presented in Figure 4.6. The inspector time for incremental scheduling is higher compared to simple software caching because of the larger volume of data that has to be hashed.

Next, the performance effects of simple software caching are quantified. The communication graph is kept constant while varying \mathcal{N}_{dup} and the volume of communication. The structure of the test loop associated with the duplicate elimination version and the pre-

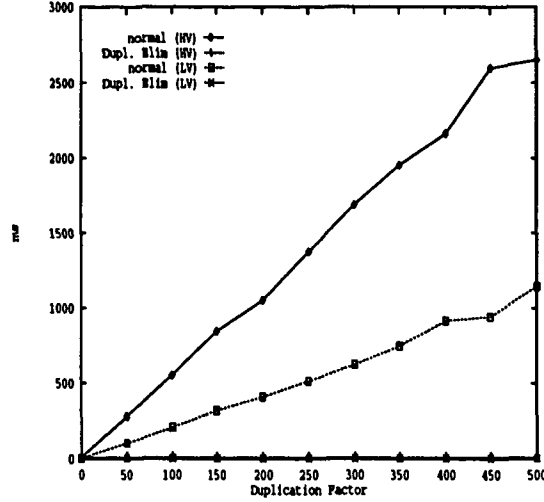


Figure 4.8: Cost vs Number of duplicates (Executor, high \mathcal{N}_{dup} , 32 Processors)

scheduled communication version is very similar to the one shown in Figure 4.3. Figure 4.7 shows the results when the duplication factor \mathcal{N}_{dup} is low, ranging from 0 to 10. Such a case is usually found in unstructured mesh computational fluid dynamic calculations. In these calculations, the connectivity of the mesh ranges from 6 to 10. Figure 4.8 shows the case where the duplication factor is very high, ranging from 0 to 500. The case is similar to the data access pattern found in molecular dynamics and particle dynamics codes, where each particle interacts with a large number of other particles (usually within a cut-off radius). Performance improvement associated with software caching increases with the duplication factor, except when the communication volume and duplication factor are both low.

Performance of the Primitives

A useful property of the workload generator is that it can be used to produce localized communication patterns whose communication structure is preserved with the increase in the number of processors. If the synthetic workload is scaled in the above manner, one cannot expect to observe significant changes in performance with increasing numbers of processors.

Table 4.5: Results Supporting Scalability (Time in secs.)

Intersection ratio	32 processors		64 processors		128 processors	
	Inspector	Executor	Inspector	Executor	Inspector	Executor
0.0	0.8	6.2	0.9	6.2	0.9	6.1
0.2	0.8	5.9	0.9	5.7	0.9	5.7
0.4	0.8	5.5	0.8	5.5	0.9	5.5

Table 4.5 illustrates the absolute timings for schedule merged incremental gathers, using the communication pattern for which the Manhattan Distance, $D = 1$, connectivity $C = 4$. The total communication volume, V , is scaled up in proportion to the number of processors employed and this maintains a volume per processor of $\approx 2K$ floating point numbers. A nearly fixed communication cost is obtained as the problem size grows linearly with the number of processors. This pattern has been observed for a number of different communication patterns. The observation supports the view that the primitives scale within the limits of scalability of the problems in which they are employed.

4.4.2 Performance Results Derived from Applications

Comparison of Communication Optimizations

A representative kernel was extracted from the Euler code and timed varying the number of processors from 16 to 128. All timings presented are for 10 iterations of the outermost loop. The communication times for the different levels of optimizations are shown in Table 4.6. It is seen that for both the 53k and 100k mesh input, schedule merging and vectorization make the communication time decrease slightly as the number of processors is increased. Similarly the total running time presented in Table 4.9, goes down significantly as more processors are used. It was shown before that if the problem is scaled as the number of processors is increased, then the primitives scale accordingly. Even though the volume of data communicated for the incremental case is the least, the buffer management to store off-processor data is complicated. Hence for certain input data, running time is higher than in other optimized cases.

Table 4.6: Euler kernel, 53k&100kMesh (Time in secs)

Optimization	Total Communication (Executor)						
	53K Mesh				100K Mesh		
	16	32	64	128	32	64	128
Simple Software Caching	22.4	22.7	29.1	37.3	29.2	29.9	34.7
Schedule Merging (SM)	19.1	20.1	24.7	28.5	25.0	25.1	26.4
Vectorized (Vect) + SM	15.9	15.7	13.1	12.8	20.7	19.3	18.1
Incremental + SM	18.9	20.2	24.3	27.9	24.3	25.1	26.7
Incremental + SM + Vect	16.1	15.7	12.9	12.7	21.2	19.1	18.0

Behavior of Paged Translation Table

Several experiments were run to measure the performance of the Paged Translation Table. Table 4.7 shows the effects of replication factor on the scheduling time for a 53k node unstructured mesh, and a benchmark input for CHARMM (MbCO + 3830 water molecules; 14026 atoms) on a 64-processor iPSC/860. The column labeled “Before” corresponds to performance with the initial block distribution of pages across the processors. The column labeled “After” corresponds to the performance after a re-organization of replicated pages, according to access behavior on each processor. In this experiment, the number of pages replicated on each processor is varied. As expected, performance improves as the replication factor increases. For the unstructured mesh, reshuffling of translation table pages does not make much difference in the scheduling time. For the molecular dynamics case, the reshuffling makes a large difference, especially for low replication factors.

Table 4.8 shows the performance of dereferences with varying block sizes for a fixed replication factor, $\mathcal{R} = 0.5$. As observed, reasonable communication times can be obtained with relatively large page sizes. When the page size is decreased, the communication efficiency of the fully replicated case can be achieved without having to replicate all the data associated with the translation table.

Table 4.7: Effects of Replication Factor (Time in secs.)

Replication	Euler kernel (53k)		CHARMM kernel	
	Before	After	Before	After
0.0	0.4	0.4	7.9	7.2
0.1	0.4	0.3	6.8	5.1
0.2	0.4	0.3	6.0	3.2
0.3	0.4	0.3	5.7	2.0
0.4	0.4	0.2	5.3	1.4
0.5	0.3	0.2	5.0	1.1
0.6	0.3	0.2	4.3	1.0
0.7	0.2	0.2	4.0	0.9
0.8	0.2	0.1	2.9	0.9
0.9	0.2	0.1	2.2	0.9
1.0	0.1	0.1	0.9	0.9

Table 4.8: Effects of Page Size, $\mathcal{R} = 0.5$, (time in secs)

Euler kernel (53k)			CHARMM kernel		
Page Size	Before	After	Block Size	Before	After
85	0.3	0.2	89	5.0	1.2
43	0.3	0.2	44	5.0	1.2
29	0.3	0.2	22	5.0	1.1
22	0.3	0.2	15	5.0	1.1
17	0.3	0.2	11	5.0	1.1
9	0.3	0.1	6	5.4	1.0
5	0.4	0.1	5	5.4	1.0
3	0.4	0.1	3	5.3	1.1

Table 4.9: Euler kernel, 53k&100kMesh (Time in secs)

Optimization	Total Running Time						
	53K Mesh				100K Mesh		
	16	32	64	128	32	64	128
Simple Software Caching	104.3	63.9	50.0	48.9	108.5	67.4	52.6
Schedule Merging (SM)	100.3	60.5	46.8	39.3	104.7	62.3	45.4
Vectorized (Vect) + SM	97.5	57.3	34.8	24.1	99.7	57.1	37.2
Incremental + SM	100.6	60.7	46.3	38.7	103.6	61.9	44.6
Incremental + SM + Vect	97.1	57.9	34.5	23.8	100.3	56.8	36.7

Table 4.10: Explicit Euler Solver Timings Using Incremental Schedule

Size Mesh	Number of Processors					
		1	2	8	16	64
3600	Mflops	4.1	7.1	16.9	17.4	-
	comp/iter(s)	4.6	2.4	0.6	0.34	-
	comm/iter(s)	-	0.25	0.48	0.73	-
26K	Mflops	-	-	23.8	38.8	
	comp/iter(s)	-	-	4.5	2.3	
	comm/iter(s)	-	-	1.1	1.1	
210K	Mflops	-	-	-	-	144.3
	comp/iter(s)	-	-	-	-	4.75
	comm/iter(s)	-	-	-	-	2.3

Performance of Optimizations on Large Scale Application

This section presents the timing results obtained from real applications that have been implemented on parallel machines using the runtime support. Timing data resulting from using both the Euler solver and the molecular dynamics code is presented.

Table 4.10 presents some timings for the explicit Euler solver [33]. These timings were obtained on the Intel Gamma machine. The multigrid Euler solver is also implemented, using the PARTI runtime support. The largest test case run so far consists of computing a highly resolved flow over a three-dimensional aircraft configuration. The mesh contains 804,056 points and approximately 4.5 million tetrahedra. The explicit unstructured mesh code achieves a rate of 1.5 Gflops on 512 Delta processors. By comparison, the unstructured solver runs at about 100 Mflops on a single processor of the CRAY-YMP, regardless of problem size for both the explicit and multigrid schemes. Similarly, both schemes achieve a computational rate of about 750 Mflops, using all eight processors of the CRAY-YMP. A well converged solution (100 multigrid cycles) can be obtained for the three-dimensional aircraft configuration in about 15 minutes on the eight processor CRAY-YMP, or just under two hours, using a single CRAY-YMP processor. When the unstructured multigrid Euler code is executed, the computational rate achieved is 1.2 Gflops on 512 Delta processors,

and the converged solution can be obtained in 10.5 minutes.

The timing data obtained for CHARMM is presented in Tables 4.11 and 4.12. Both an irregular block partitioning (with load balancing) and a recursive coordinate bisection scheme were used to partition the data. The largest input file consisted of 14026 atoms ((MbCO + 3830 water molecules). The timings obtained are comparable to all other implementations [20]. From the results, it is clear that the implementation, which uses binary dissection to partition the data scales better than the blocked partition implementation. Experiments in which the partitioning was performed based on geometry and the workload on each atom yielded the best results. The indirection array generated for the force calculation has an extremely high duplication factor. Each atom in the calculation interacts with hundreds of other atoms, hence the large duplication factor.

Table 4.11: CHARMM timings using Irregular Block Data Partition

Nodes	E _{ext} ¹	E _{int} ²	Comm ³	List ⁴	Total	Eff. ⁵	Speedup
1	7023.9	44.5	0.0	382.1	7459.5	100.0%	1
16	421.57	2.65	43.17	17.50	486.85	95.7%	15
32	212.47	1.34	44.68	9.06	268.43	86.8%	28
64	108.11	0.69	52.62	4.85	167.15	70.0%	45
128	53.43	0.35	62.33	2.78	119.22	48.8%	63

¹ Nonbond energy: Electrostatic, van der Waals

² Internal energy: Bond, Angle, Torsion,...

³ Total communication times

⁴ Nonbond list generation times

⁵ Efficiency for N processors is defined by the following ratio: $\frac{(\text{time for 1 proc})}{(\text{time for N processors}) \times N}$

Table 4.12: CHARMM timings using Binary Dissection Partition

Nodes	E _{ext} ¹	E _{int} ²	Comm ³	List ⁴	Total	Eff. ⁵	Speedup
1	7023.9	44.5	0.0	382.1	7459.5	100.0%	1
16	465.98	2.42	22.92	18.55	511.15	91.2%	15
32	294.85	1.21	24.58	10.08	331.42	70.3%	23
64	194.61	0.64	23.82	5.66	225.83	51.6%	33
128	101.70	1.14	25.79	3.20	132.13	44.1%	57

Chapter 5

Loop Transformations

An increasing fraction of the applications targeted by parallel computers make heavy use of indirection arrays for indexing data arrays. A limitation of existing techniques addressing this problem is that they are only applicable for a single level of indirection. However, many codes using sparse data structures access their data through *multiple levels of indirection*. A number of compilers have implemented the inspector/executor transformation of an irregular loop, so that it can be executed on a distributed memory machine. Other than the author's implementation of the inspector/executor transformation the Kali compiler [60] and the Vienna Fortran compiler [18] have also successfully implemented this transformation.

This section presents a method for transforming programs using multiple levels of indirection into programs with, at most, one level of indirection, thereby broadening the range of applications that a compiler can parallelize efficiently. A central concept of this algorithm is to perform *program slicing* on the subscript expressions of the indirect array accesses. Such slices peel off the levels of indirection, one by one, and create opportunities for aggregated data prefetching in between. A *slice graph* eliminates redundant preprocessing and gives an ordering in which to compute the slices. The work is presented in the context of High Performance Fortran.

HPF offers the promise of significantly easing the task of programming distributed mem-

ory machines and making programs independent of a single machine architecture. Current prototypes of compilers for HPF-like languages produce Single Program Multiple Data (SPMD) code with message passing and/or runtime communication primitives.

Reducing communication costs is crucial in achieving good performance on applications [51, 53]. While current systems like the Fortran D project [54] and the Vienna Fortran Compilation system [22] have implemented a number of optimizations for reducing communication costs (like message blocking, collective communication, message coalescing and aggregation), these optimizations have been developed mostly in the context of regular problems (*i.e.*, for codes having only regular data access patterns). Special effort is required in developing compiler and runtime support for applications that do not have regular data access patterns.

When irregular loops are parallelized, the off-processor data must be pre-fetched before the loop computation begins. If the off-processor data is not pre-fetched, data communication inside the computation loop will occur, resulting in bad performance. Runtime support, analysis techniques, and compiler prototypes have been designed to transform loops where distributed arrays are accessed through a single level of indirection into inspector/executor pairs. During program execution, the inspector examines the data references made by a processor and calculates what off-processor data need to be fetched and where to store it once received. The executor loop then uses the information from the inspector to implement the actual computation.

An example for the class of kernels that can be handled by the techniques, developed so far, is the irregular kernel in Figure 5.1. In this example, data arrays *col*, *x* and *y* are block distributed between processors. The *i*-loop iterations are partitioned using the HPF-directive **ON_HOME**, which in this case is equivalent to the *owner computes rule* that assigns the computation of an assignment statement to the processor that stores the left-hand side reference. A single level of indirection arises because data array *y* is indexed, using the array *col* in statement K2.

While such simple indirection patterns can be handled, many application codes have

```

SUBROUTINE simple(x, y, col, m, n)

  INTEGER i, m, n, col(m)
  REAL x(n), y(n)
  !HPF$ DISTRIBUTE(BLOCK) :: col, x, y

  !HPF$ EXECUTE (i) ON_HOME x(i)
K1      FORALL i = 1, n
K2      x(i) = x(i) + y(col(i))
K3      ENDFORALL
K4      END

```

Figure 5.1: Kernel with single level of indirection.

code segments and loops with more complex access functions that go beyond the scope of current compiling techniques. In many cases, a chain of distributed array indexing is set up where values stored in one distributed array are used to determine the indexing pattern of another distributed array, which in turn determines the indexing pattern of a third distributed array. Such loops with multiple levels of indirection are very common and appear, for example, in unstructured and adaptive applications codes associated with particle methods, molecular dynamics, sparse linear solvers and, in some, unstructured mesh CFD solvers.

This section develops techniques that can be used by compilers to transform loops with array accesses, involving more than a single level of indirection into loops where array references are made through, at most, one level of indirection. This transformation technique is presented in the context of distributed memory machines and therefore often refers to prefetching as “communication” or “message blocking.” However, this method is likely to be useful on any architecture where it is profitable to prefetch data between different levels of a memory hierarchy.

The rest of this section is organized as follows: Section 5.1 gives an overview of the transformation technique by transforming an example code that shows two levels of indirection. Section 5.3 introduces some terminology that is used in Section 5.4, which gives

```

SUBROUTINE CSR(x, y, col, ija, m, n)

  INTEGER i, j, m, n, col(m), ija(n)
  REAL x(n), y(n)
  !HPF$DISTRIBUTE(BLOCK) :: col, ija, x, y

  !HPF$EXECUTE (i) ON_HOME x(i)
R1      FORALL i = 1, n
R2          x(i) = 0
R3          DO j = ija(i) + 1, ija(i + 1)
R4              x(i) = x(i) + y(col(j))
R5          ENDDO
R6      ENDFORALL
R7      END

```

Figure 5.2: CSR kernel – original version.

a formal description of the algorithms and illustrates how the transformation, shown in Section 5.1, was derived. Section 5.4.6 concludes with a brief discussion on how to use incremental scheduling.

5.1 Example Transformation

This section illustrates the effect of applying the transformation to the HPF subroutine *CSR*, shown in Figure 5.2. The code is based on a sparse matrix vector multiplication kernel and uses the Compressed Sparse Row format [92]. The matrix values are all assumed to be equal to zero or one. The columns associated with non-zero entries in row i are specified by $col(j)$, where $ija(i) + 1 \leq j \leq ija(i + 1)$. For simplicity, all distributed arrays are distributed blockwise in this example; these techniques apply equally well to other potentially irregular decompositions. The indexing of y by array col causes a first level of indirection. The dependence of the loop bounds of the inner j -loop on the distributed array ija causes an additional level of indirection. This double indirection becomes clear when

rewriting the computation as

$$x(i) = \sum y(col(ija(i) + 1 : ija(i + 1)))$$

for $i = 1 \dots n$.

All references to the distributed array x are indexed by the loop induction variable i . The HPF `ON_HOME` construct partitions the iteration space of the `FORALL` loop so that iteration i is performed on the processor that owns $x(i)$; there is no communication required for referencing x . For the other three arrays, ija , col and y , data communication is required. As already mentioned, keeping the total number of these communication steps down is key to high performance on a distributed memory machine. Therefore, only a small number of aggregate prefetch operations should be performed, instead of communicating each reference individually. This operation requires a significant amount of preprocessing to determine what data need to be prefetched and in which order. The code will be transformed so that the compiler runtime support will have access to the subscripts of all elements of ija , col and y that need to be prefetched from other processors. This information makes it possible to carry out the communication optimizations described previously, *i.e.*, to reduce the volume of communication, reduce the number of messages and to prefetch off-processor data to hide communication latencies.

The transformed version of subroutine *CSR* is shown in Figures 5.3 and 5.4. For ease of presentation, a variation of HPF that contains additional directives `BEGIN LOCAL` and `END LOCAL` is used to indicate *local variables*. These variables do not reside in the global name space inhabited by the other HPF variables, but instead they exist independently in the local name space of each processor. In strict HPF, such variables can be emulated by either adding another dimension of size $n\$proc$ (the total number of processors) and referencing this dimension with $my\$proc$ (the id of each processor) or by manipulating them only through so called extrinsic functions. Except for these local variables, the whole code is presented in global name space, and for simplicity, it is assumed that all global to

```

SUBROUTINE CSR(x, y, col, ija, m, n)

  INTEGER i, j, m, n, col(m), ija(n)
  REAL x(n), y(n)
  !HPF$ DISTRIBUTE(BLOCK) :: col, ija, x, y

  !HPF$ BEGIN LOCAL
    INTEGER v4, v5
    INTEGER, ALLOCATABLE(:) ::
      . v1arr, v2arr, v3arr
  !HPF$ END LOCAL

  C    COUNTING SLICE D
  C    Count local iterations of outer loop
  C    to determine size of v1arr.
T1    v4 = 0
T2 !HPF$ EXECUTE (i) ON_HOME x(i)
T3    FORALL i = 1, n
T4      v4 = v4 + 1
T5    ENDFORALL

  C    COLLECTING SLICE A
  C    Collect "i + 1" into v1arr(1:v4).
S1    ALLOCATE (v1arr, v4)
S2    v4 = 0
S3 !HPF$ EXECUTE (i) ON_HOME x(i)
S4    FORALL i = 1, n
S5      v4 = v4 + 1
S6      v1arr(v4) = i + 1
S7    ENDFORALL
S8 C    Prefetching ija(v1arr(1:v4)) goes here

  C    COUNTING SLICE E
  C    Count local iterations of inner loop to
  C    determine size of v2arr and v3arr.
T6    v4 = 0
T7    v5 = 0
T8 !HPF$ EXECUTE (i) ON_HOME x(i)
T9    FORALL i = 1, n
T10     v4 = v4 + 1
T11     DO j = ija(i) + 1, ija(v1arr(v4))
T12       v5 = v5 + 1
T13     ENDDO
T14    ENDFORALL

```

Figure 5.3: CSR kernel – transformed version (Part 1).

```

C      COLLECTING SLICE B
C      Collect "j" into v2arr(1:v5).
S9      ALLOCATE (v2arr, v5)
S10     v4 = 0
S11     v5 = 0
S12 !HPF$ EXECUTE (i) ON_HOME x(i)
S13     FORALL i = 1, n
S14         v4 = v4 + 1
S15         DO j = ija(i) + 1, ija(v1arr(v4))
S16             v5 = v5 + 1
S17             v2arr(v5) = j
S18         ENDDO
S19     ENDFORALL
S20 C    Prefetching col(v2arr(1:v5)) goes here

C      COLLECTING SLICE C
C      Collect "col(j)" into v3arr(1:v5).
S21     ALLOCATE (v3arr, v5)
S22     v4 = 0
S23     v5 = 0
S24 !HPF$ EXECUTE (i) ON_HOME x(i)
S25     FORALL i = 1, n
S26         v4 = v4 + 1
S27         DO j = ija(i) + 1, ija(v1arr(v4))
S28             v3arr(v5) = col(v2arr(v5))
S29             v5 = v5 + 1
S30         ENDDO
S31     ENDFORALL
S32 C    Prefetching y(v3arr(1:v5)) goes here

C      ACTUAL COMPUTATION
E1      v4 = 0
E2      v5 = 0
E3 !HPF$ EXECUTE (i) ON_HOME x(i)
E4      FORALL i = 1, n
E5          x(i) = 0
E6          v4 = v4 + 1
E7          DO j = ija(i) + 1, ija(v1arr(v4))
E8              v5 = v5 + 1
E9              x(i) = x(i) + y(v3arr(v5))
E10         ENDDO
E11     ENDFORALL
E12     END

```

Figure 5.4: CSR kernel – transformed version (Part 2).

local address translations will be handled by the HPF compiler. Note, however, that index translation in the presence of indirect addressing and further complications, like irregular decompositions, is a nontrivial task; the code actually generated by this implementation assists in the address translation process.

In the example, the distributed array ija is distributed conformable to the array x ($ija(k)$ is always assigned to the same processor as $x(k)$). Since the reference $ija(i)$ in statement R3 occurs in a FORALL loop whose iteration space is aligned to the index space of x , this reference does not generate any communication. It is also assumed that the back end compiler recognizes the use of induction variable i in this reference and does not require any preprocessing for performing the global to local name space conversion.

The references $ija(i + 1)$, $col(j)$, and $y(col(j))$, however, may require preprocessing. In general, for a reference of the form $arr(sub_{ast})$, the preprocessing may perform the following:

- It must collect all values of sub_{ast} used by a processor in order to prefetch the data referenced in $arr(sub_{ast})$ en bloc. In some cases, preprocessing is also carried out to reduce communication volume through recognition of duplicate references in sub_{ast} .
- It has to provide a mechanism to access the prefetched data during the actual computation.

Here, sub_{ast} stands for the Abstract Syntax Tree (AST) index of the subscript. Note that while this index is different for each reference in the program, the value numbers of these references may be identical, even for subscripts that might textually appear different.

In the transformed code, the statements proceeding the actual computation (in E1...E12) perform this preprocessing. Statements S8, S20 and S32 indicate opportunities for aggregated prefetching of the data required for references $ija(i + 1)$, $col(j)$ and $y(col(j))$, respectively. For the *CSR* kernel, it is assumed that subscript reuse is relatively low. Therefore, the prefetching and indexing are performed via temporary trace arrays that store global indices and are themselves indexed through counters that are incremented with each reference. Alternative mechanisms are described in Section 5.4.2.

The first prefetch statement, S8, brings in the trace of the reference $ija(i + 1)$. Statements T1...T5 and S1...S7 perform the preprocessing necessary for the prefetch. Since this example is basing the prefetching mechanism on temporary trace arrays that have to be allocated dynamically, the size of the trace, *i.e.*, the number of references has to be determined first. This size is computed and stored in v_4 by statements T1...T5. Statement S1, then, allocates the local array $v1arr$, which has been declared ALLOCATABLE. Statements S2...S7 generate and store the trace into $v1arr$. Finally, the prefetching operation in S8 brings in all the non-local data and stores them in the right locations of the array ija . This process might require resizing the array ija to store the off-processor data. For the purpose of this example, it is assumed that storing of the off-processor data in the resized ija array is such that they can be referenced in global coordinates.

The next potential communication is generated by the prefetching statement, S20, which collects on each processor the off-processor references to $col(j)$ in statement R4. Statements S10...S19 collect the trace of the value j indexing the array col into the local array $v2arr$. Note that in the expression for the upper bound of the j -loop, array ija is no longer indexed by $(i + 1)$ but by the trace vector $v1arr$ generated in statements S4...S7. The statements T6...T14 in Figure 5.3 compute the size of the array $v2arr$ into the local scalar v_5 . The array $v2arr$, like $v1arr$ has been declared ALLOCATABLE in statement S9.

The values of y that are required on each processor at statement R4 are communicated in the prefetching statement S32. The trace of the values that index y is done in statements S22...S31 and it is stored in the dynamic local array $v3arr$. Note that the number of references to $y(col(j))$ is the same as the number of references to $col(j)$; therefore, the size of $v3arr$ is the same as the size of $v2arr$. Hence there is no need for any additional code to find out the size of $v3arr$; instead, the already computed local variable v_5 that stores the size of $v2arr$ can be used. Note also that in statement S28 the array col is referenced by the local array $v2arr$, which stores global indices, instead of being referenced by j . After the execution of statement S32, all processors have the required values of y in their local memories.

The actual loop computation is performed in statements E1...E11. During this computation, no communication is required because everything that is necessary on each processor has already been fetched. To summarize, the original code shown in Figure 5.2 has been transformed into the code in Figures 5.3 and 5.4 and the transformed code does all the necessary data communication in phases after several preprocessing steps. Within the different loops in the transformed code, all distributed arrays are referenced by at most one level of indirection and require no data communication.

In the CSR kernel example, there is no assignment to the indirection array. If there is an assignment inside the compute loop, the method will work and this process will be explained when the slice generation process is described. The method suggested here is completely general and will work for all cases though it might not produce the most efficient code when an assignment to the indirection arrays exists at the innermost loop.

If the program CSR_init was executed on a shared memory multiprocessor which has no memory hierarchy (*i.e.*, there is only one main memory and memory access time to read consecutive words, is the same as the time required to read two words at arbitrary locations), then the transformation presented here becomes redundant. But for all real machines this is not the case, hence such a transformation which prefetches data into contiguous locations helps to speed up the computation.

5.2 Preliminaries

In this section, some of the compiler terminology that will be used in the description of the algorithm is clarified.

Abstract Syntax Tree: After the parser analyzes the program, it maps it onto a tree structure called the Abstract Syntax Tree. The program analysis and transformation is done on the AST, and this goes into the backend of the compiler. An AST node is generated for every basic element of the input code.

Value Number: An abstract value graph is generated for the code that is being parallelized. Value numbering is a symbolic analysis tool that can be effectively used to do subexpression elimination during code optimization. A value is given to each node of the abstract syntax tree. Analysis is done so that, if it can be guaranteed that two variables (syntactically different) will have the same value during program execution, they will have the same value number in the value graph.

```

S1   A = 5
S2   B = 5
S3   C = A + B
S4   D = B + A

```

In the value graph for the above piece of code variables C and D will have the same value number, even though they are syntactically different.

5.3 Definitions

This section introduces some concepts that will be used in the algorithms in Section 5.4.

A *Slice* is a tuple

$$s = (s_{vn}, s_{target}, s_{code}, s_{ident}, s_{dep_set}, s_{cnt_vn})$$

that contains a value number s_{vn} , a designated program target location s_{target} , a sequence of statements s_{code} , an identifier s_{ident} , a dependence set s_{dep_set} , and optionally another value number s_{cnt_vn} . There are two types of slices:

- A *collecting slice* stores the sequence of values (trace) that are assigned to a variable (e.g. Figure 5.2: statement R3 reference $i + 1$; statement R4 reference $col(j)$) during the execution of the program in some data structure identified by s_{ident} . The type of the data structure is determined by the degree of subscript reuse within the trace of the subscript, as described in Section 5.4.2. Examples of collecting slices are shown in

Figure 5.4. Two slices B and C , shown in statements S9-S19 and S21-S31 respectively, are collecting slices.

- *Counting slices* are created from the collecting slices; they calculate the size of the subscript trace that will be generated during the execution of the collecting slice. A counting slice is needed if the collecting slice requires the size of the trace it is to record (for example, preallocating a data structure to store the trace). Examples of counting slices are shown in Figure 5.3. The two slices, D and E shown in statements T1-T5 and T6-T14 respectively, are counting slices.

Each of the slices has the following properties with respect to the original program P :

- Inserting s_{code} at s_{target} in P is legal; i.e., it does not change the meaning of P . The s_{code} is similar to a dynamic backward executable slice [99].
- After executing s_{code} , s_{ident} will have stored the values of s_{vn} .
- If s is a *collecting slice*, then s_{vn} will be the value number of a subscript sub_{ast} of a nonlocal array reference $arr(sub_{ast})$ in P , and s_{ident} will store the sequence of all the values that sub_{ast} will be assigned during the execution of P . Note that the length of this sequence depends on the location in the program, which is given by s_{target} . For example, if s_{target} is the statement of the reference itself, then the sequence consists of only a single subscript. If s_{target} is the header of a loop enclosing the reference, then the sequence contains the subscripts for all iterations of the loop.
- If counting slices are computed, then s_{cnt_vn} will be the value number of the counter indexing s_{ident} after execution of s_{code} is finished; i.e., the value of s_{cnt_vn} will be the size of the subscript trace computed in s_{ident} .
- If s is a *counting slice*, then there exists a collecting slice t for which $s_{vn} = t_{cnt_vn}$ and $s_{target} = t_{target}$. s_{ident} will store the size of the subscript trace computed in t_{ident} . Since s_{ident} corresponds to a single value, s_{cnt_vn} will be the value number corresponding to the constant “1.” Note: $s_{target} = t_{target}$ because otherwise too many

(for s_{target} preceding t_{target}) or too few (for s_{target} succeeding t_{target}) subscripts may be counted.

- The s_{dep_set} stored in each slice is a set of AST indices of subscript variables that need runtime processing. Only the references in s_{code} that require runtime processing are considered when the s_{dep_set} is created.

A *Slice Graph* is a directed acyclic graph

$$G = (S, E)$$

that consists of a set of slices S and a set of edges E . For $s, t \in S$, an edge $e = (s, t) \in E$ establishes an ordering between s and t . The presence of e implies that t_{code} contains a direct or indirect reference to s_{ident} and therefore has to be executed after s_{code} . G has to be acyclic to be a valid slice graph. Note that the edges in the slice graph not only indicate a valid ordering of the slices, but they also provide information for later optimizations. For example, it might be profitable to perform *loop fusion* across slices; the existence of an edge between slice nodes, however, indicates that these slices cannot be fused.

A *Subscript Descriptor*

$$sub_{ast} = (sub_{vn}, sub_{target})$$

for the subscript sub_{ast} of some distributed array reference consists of the value number of sub_{ast} , sub_{vn} and the location in P , where a slice generated for sub should be placed, sub_{target} . The algorithm will generate a slice for each unique subscript descriptor corresponding to a distributed array reference requiring runtime preprocessing. Identifying slices by subscript descriptors is efficient in that it allows a slice to be reused for several references, possibly of different data arrays, as long as the subscripts have the same value number. It is conservative in that it accounts for situations where different references might have the same subscript value number but different constraints with respect to prefetch aggregation that corresponds to different target nodes.

5.4 The Algorithm

This section gives a description of the algorithm to perform the transformation shown in Section 5.1. The algorithm consists of two parts: The first part, described in Section 5.4.1, analyzes the program and generates the slices and the slice graph. The second part, described in Section 5.4.5, uses the slice graph to do the code generation.

5.4.1 Slice Graph Construction

The procedure **Generate_slice_graph()**, shown in Figure 5.5, is called with the program P and the set of subscripts R of the references that need runtime preprocessing, *i.e.*, the irregular references. It returns a slice graph consisting of a set of slices S and edges E . This procedure first generates all the necessary slices and then finds the edges between these slices.

The Foreach statement in A4...A8 computes a subscript descriptor (sub_{vn}, sub_{target}) for each subscript AST index sub_{ast} . It is assumed that P has an associated value number table that maps AST indices to value numbers. **Lookup_val_number()** uses this table to compute sub_{vn} from sub_{ast} . **Gen_target()** maps the AST index sub_{ast} to the target node sub_{target} for the slice generated, starting from that AST index. The constraints on sub_{target} are the following:

- In the Control Flow Graph (CFG), sub_{target} predominates the reference sub_{ast} ; *i.e.*, it is guaranteed that sub_{target} will be executed before sub_{ast} is used to reference its data array arr .
- There is no modification of the data array arr between sub_{target} and sub_{ast} .
- Any code inserted at sub_{target} is executed as infrequently as possible.

Gen_target() implements these constraints using a Tarjan interval tree [1] and array MOD information; starting at the node corresponding to the reference, it walks the interval tree upwards and backwards until it reaches a modification of arr .

```

Procedure Generate_slice_graph( $P, R$ )

  //  $P$ : Program to be transformed
  //  $R$ : AST indices of subscripts of references
  //      that need runtime preprocessing

  A1  $S := \emptyset$  // Slices
  A2  $E := \emptyset$  // Slice ordering edges
  A3  $U := \emptyset$  // Subscript descriptors

  // Compute subscript descriptors.
  A4 Foreach  $sub_{ast} \in R$ 
  A5    $sub_{vn} := \text{Lookup\_val\_number}(sub_{ast})$ 
  A6    $sub_{target} := \text{Gen\_target}(sub_{ast})$ 
  A7    $U := U \cup \{(sub_{vn}, sub_{target})\}$ 
  A8 Endforeach

  // Compute slices.
  A9 Foreach  $sub \in U$ 
  A10   $s := \text{Gen\_slice}(sub)$ 
  A11   $S := S \cup \{s\}$ 

  // The following steps are executed
  // iff counting slices are required.
  O1   $t := \text{Lookup\_slice}(S, (s_{cnt\_vn}, s_{target}))$ 
  O2  If  $t = \emptyset$  Then
  O3     $t := \text{Gen\_slice}(s_{cnt\_vn}, s_{target})$ 
  O4     $S := S \cup \{t\}$ 
  O5     $E := E \cup \{(t, s)\}$ 
  O6  Endif

  A12 Endif
  A13 Endforeach

  // Compute edges resulting from
  // dependence sets of slices.
  A14 Foreach  $s \in S$ 
  A15   Foreach  $sub_{ast} \in s_{dep\_set}$ 
  A16     $sub_{vn} := \text{Lookup\_val\_number}(sub_{ast})$ 
  A17     $sub_{target} := \text{Lookup\_target}(sub_{ast})$ 
  A18     $t := \text{Lookup\_slice}(S, (sub_{vn}, sub_{target}))$ 
  A19     $E := E \cup \{(t, s)\}$ 
  A20   Endforeach
  A21 Endforeach

  A22 Return ( $S, E$ )

```

Figure 5.5: Slice graph generation algorithm.

The next Foreach statement in A9...A13 iterates through the subscript descriptors $sub \in U$ and generates for each subscript descriptor both the collecting slice s and, if needed, the counting slice t . **Gen_slice()** takes a subscript descriptor $sub = (sub_{vn}, sub_{target})$ and generates for location sub_{target} the slice that computes the values corresponding to sub_{vn} . The slice generation function uses the program's CFG and the SSA (Static Single Assignment). Roughly speaking, **Gen_slice()** follows the use-def and control dependence chain starting in sub_{ast} until it reaches sub_{target} .

If the size of the subscript trace recorded in s (e.g., for allocating trace arrays) is required, then the statements O1...O6, a counting slice t for each s , is executed. However, different collecting slices can share a counting slice, if they have the same counter value number sub_{cnt_vn} and target location sub_{target} . Therefore, the set of already created slices must first be examined. **Lookup_slice()** takes as input a set of slices S and a subscript descriptor sub , and returns the slice $t \in S$ corresponding to sub if there exists such a t ; otherwise, it returns \emptyset . If a counting slice has not yet been created, a new counting slice t is generated. Since the counting slice t must be executed before the collecting slice s , a directed edge (t, s) is added to the edge set E .

The nested Foreach statements in A14...A21 are used to find the directed edges resulting from the dependence sets in each slice. The outer Foreach iterates through the slices s and the inner one loops through the references sub_{ref} stored in the dependence set s_{dep_set} of s . All the relevant information has already been generated previously; therefore, these loops only have to consult tables to complete the set of edges.

The slice graph corresponding to the transformation example, done in Section 5.1, is shown in Figure 5.6. There are five nodes in the slice graph, of which nodes A, B and C contain collecting slices, while nodes D and E contain counting slices. Note that the collecting slices B and C share the counting slice E, which reflects that the number of references to $y(col(j))$ is the same as the number of references to $col(j)$.

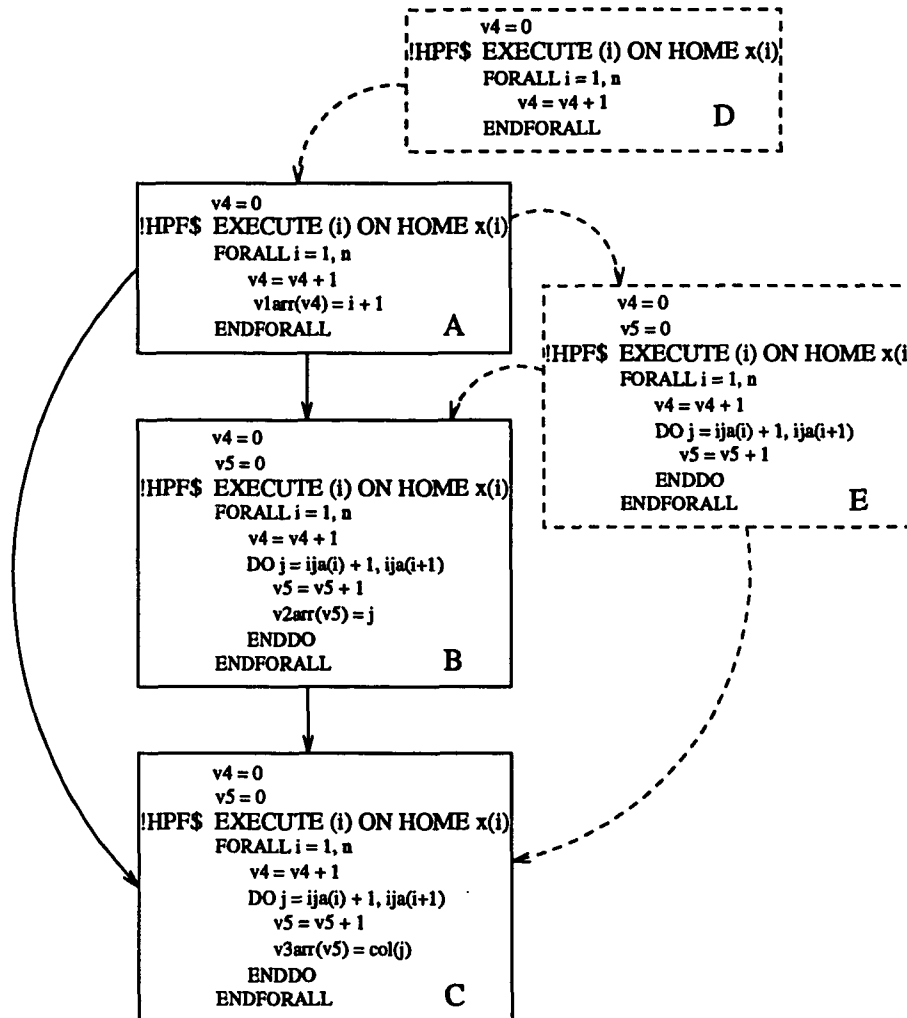


Figure 5.6: Example of a Slice Graph.

5.4.2 Trace Management Schemes

Precomputing the subscript trace has been defined so that prefetching can be performed. Before actually generating code, however, decision has to be made regarding the data structures to be used for first recording the traces to prefetch nonlocal data and then accessing these prefetched data. The example presented in Section 5.1 used temporary trace arrays for performing both of these operations. It turns out, however, that this is just one of several options, and there are different tradeoffs involved depending on the characteristics of the

subscript traces. Consequently, when generating the statements s_{code} of a slice s , the code for manipulating these data structures is not included, *i.e.*, the counter initializations and increments or the assignments into trace arrays. Instead, place holders for these operations are included and the generation of these statements are delayed until the slice instantiation phase.

Let T be the size of the trace, *i.e.*, the number of times a subscript is evaluated with respect to the target location of the slice; let R be the number of unique elements in T , and let N be the global size of the subscripted array, *i.e.*, the number of different subscripts possible. Note that $R \leq N$, $R \leq T$ must hold.

5.4.3 Case 1: Low subscript reuse

In this case, which is characterized by $R \approx T$, each subscript typically appears at most once in the trace produced by the slice. A possible example is the CSR kernel described in Section 5.1. Here it is reasonable to use a *dynamically allocated array* that is indexed through a counter incremented with each reference. This array can be used both for precomputing the subscripts and for looking them up during the actual computation. Since each subscript must be stored individually, the space requirements are $\mathcal{O}(T)$. Usually counting slices must be generated to perform the dynamic allocation of the arrays. The time per access, however, is only $\mathcal{O}(1)$.

5.4.4 Case 2: High subscript reuse

This case is characterized by $R < T$; each subscript typically appears several times in the trace produced by the slice. An example of this is the pair list used for the non-bonded force kernel in molecular dynamics applications. Since each atom interacts with many other atoms, it appears many times in the pair list. Here some set representation, like a hash table, which collects subscripts and stores each of them at most once, would be an appropriate trace recording mechanism. Using a hash table to store off-processor data values was first introduced in [52]. The space requirements are only $\mathcal{O}(R)$, and counting

slices are not needed. The time per access, however, will be $\mathcal{O}(\log(R))$ for most common set representations.

As a subscripting mechanism in the actual computation, some dictionary representation, can be used, like a hash table (of a different kind than the one used for representing sets), that maps global indices to local indices. This typically requires space $\mathcal{O}(N)$ and $\mathcal{O}(\log(N))$ time per access.

An alternative subscripting mechanism is a “global shuffle,” where, roughly speaking, everything is translated to local coordinates, including the subscripting arrays themselves. The space requirements would be at most $\mathcal{O}(N)$, depending on how much data a processor needs locally and whether things can be shuffled in place or not. The time per access would be $\mathcal{O}(1)$.

5.4.5 Code generation

The code generation algorithm is shown in Figure 5.7. The procedure **Gen_code()** takes as input the original program P and the slice graph consisting of slices S and their ordering E . **Gen_code()** traverses the program and changes the subscripts of all the references that required runtime preprocessing. The function **Instantiate_program()** takes the program P and the set of slices S and replaces the subscripts in P on which preprocessing has been performed, with accesses to data structures defined in the preprocessing phase. The program instantiation depends on what type of data structure is used to store the trace of subscripts in the collecting slices, as discussed in Section 5.4.2.

Topological_sort() performs a topological sort of the slice graph, so that the partial order given by the directed edges in E is maintained during generating code for the slices in S . The **Foreach** statement in C3...C6 iterates through the slices S . **Instantiate_slice()** is similar to **Instantiate_program()**, but instead of a program P , it takes a slice s . However, it not only replaces subscript references but also adds the code mentioned in Section 5.4.2 for collecting the subscript trace. Accordingly, this instantiation, like the program instantiation, depends on the type of data structure that is used to store the subscript trace of

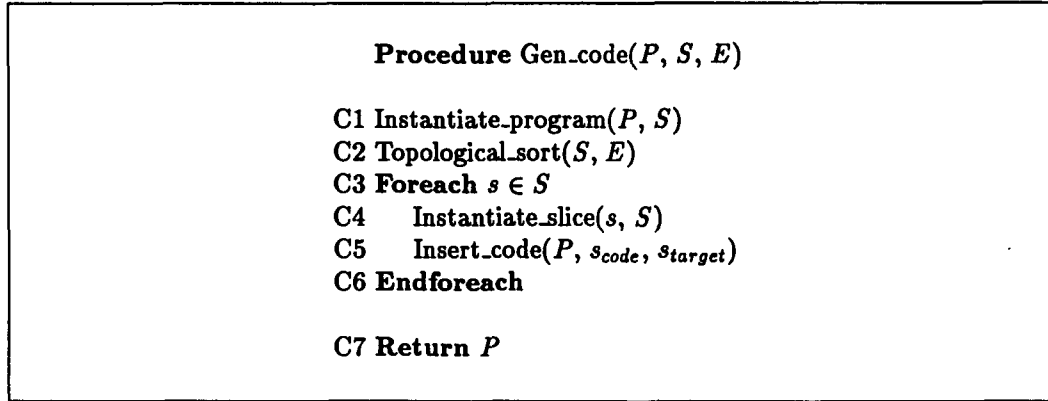


Figure 5.7: Code generation algorithm.

the references that affect the computation in this slice. After s has been instantiated, **Insert_code()** inserts s_{code} into the program at the target location s_{target} . The transformed program is returned to the calling procedure.

In the CSR example in Section 5.1, it is assumed that the subscript traces are stored in dynamically allocatable arrays. The instantiation routines add the code for maintaining and referencing these arrays to the slices in the graph presented in Figure 5.6. A topological sort on the graph yields the node order to be D, A, E, B and C; this is the same order in which the slices appear in the transformed code in Figure 5.5. For each of the slices, the subscripts of the references requiring runtime preprocessing present in the slice are changed to the local array that stores a trace of the subscript. At runtime the trace must already have been generated because an edge from the node exists where the trace was created to the node where it is being used. The slice is substituted in the program before the slice target node. Note that the topological sort order is unique; this indicates, for instance, that there is no loop fusion possible in the example. Note also that the transformed code in Figures 5.3 and 5.4 would be equally valid without having the subscripts of the references $ija(i+1)$, $col(j)$ and $y(col(j))$ replaced with references to trace arrays. However, this replacement makes the subsequent task of translating global indices to local indices simpler; instead of having to modify user declared variables and subscript arrays, it is sufficient to translate the trace arrays.

5.4.6 Using Incremental Scheduling

The use of incremental schedules makes it possible to avoid retransmission of unchanged distributed array references. Proper use of incremental schedules can have a marked effect on the communication time. The generation of incremental schedules can be carried out in two passes. A compiler first generates an inspector and executor for loop L with full schedules. During the second pass, some full schedules are replaced with incremental schedules.

Substantial analysis must be carried out if incremental schedules are used to eliminate duplicate data communication between loops. For this, comprehensive information about the program behavior is required. To use incremental scheduling, the following must be known:

- when off-processor data copies become invalidated by new assignments, and
- which communication schedules have been already invoked by the time one reaches a distributed array reference.

Such information will be available if one performs a global data flow analysis. Global dataflow analysis has been investigated for the purpose of incremental scheduling together with researchers from Rice University [46].

Chapter 6

Implementation Issues

This chapter covers some of the details about the loop transformation implementation accomplished by using the infrastructure developed for the Fortran D compiler project, at Rice University. An implementation of the transformation algorithm presented in Chapter 5 has been completed and further improvements are being carried out.

The Fortran D compiler environment has been chosen for implementation of the transformations because of the availability of various symbolic analysis tools. A brief description of these tools and how they were utilized to perform the transformations is included in this chapter, which is divided into three sections. Section 6.1 covers the symbolic analysis tools, followed by the section describing program slicing and how it is utilized to perform inspector generation. The last section gives a high-level description of the implementation already completed.

6.1 Symbolic Analysis

Symbolic analysis helps to perform various types of code transformations to vectorize or parallelize a given code. It is a powerful analysis tool that allows one to perform various code optimizations [1], such as common subexpression elimination, detection of loop invariant computation, code motion to move invariant to preheader of loop, induction variable elimination, etc. This section offers a brief description of the various symbolic analysis tools

utilized in this work.

Two types of dependencies exist in programs: *data dependence* and *control dependence*. These dependencies are best explained by using examples.

```
S1    A = B + C
S2    D = C * A
```

The execution order of these two statements has an effect on the calculated value of the variable D. Switching the order of the statements will give variable D an incorrect result. In such a case, a data dependence exists between statements S1 and S2. Data dependencies can be further subdivided: *true*, *anti* and *output* dependencies [4].

```
S1    if (A) then
S2      B = C + D
S3    end if
```

In the above case, the value of variable A decides whether statement S2 will be executed or skipped. In such cases, there exists a control dependence between statements S1 and S2. However, control dependence can always be replaced by an equivalent data dependence [4].

The control flow graph (CFG) is a DAG that represents the flow of control between the *basic blocks* of a program. A basic block is a sequence of statements with a single entry pad (first statement) and a single exit pad (last statement). Branching statements cannot be present in the basic block. In the representation of CFG used here, a graph node is generated for each basic block that may contain zero or more statements. There is a special *ENTRY* node that has no incoming edges but one or more outgoing edges to each entry point of the program. There also exists a special *EXIT* node that does not have any out edges but has a number of incoming edges from each exit point of the program. For any node b_i in this graph, there exists a path from *ENTRY* to b_i and a path from b_i to *EXIT*. Hence,

$$CFG = (V, E),$$

S1	a = 1	T1	a ₁ = 1
S2	b = 2	T2	b ₁ = 2
S3	if (a) then	T3	if (a ₁) then
S4	b = b + 2	T4	b ₂ = b ₁ + 2
S5	end if	T5	end if
S6	c = b + d	T6	b ₃ = $\phi(b_1, b_2)$
S7	a = a - c	T7	c ₁ = b ₃ + d ₁
		T8	a ₂ = a ₂ - c ₁

Figure 6.1: Program fragment and SSA form

where $V = \{b_1, b_2, \dots, b_n, ENTRY, EXIT\}$, b_1, b_2, \dots, b_n represent the nodes corresponding to the basic blocks and E is the set of edges. For $b_i, b_j \in V$, an edge $e = (b_i, b_j) \in E$, establishes a flow of control from block b_i to block b_j .

The program dependence graph (*pdg*) [38] provides an intermediate representation of the program. Each statement in the program is a node in the *pdg*. When there is an edge from node A to node B, there exists either a control dependence or a data dependence between the statement represented by node A and the statement symbolized by node B.

The static single assignment (SSA) form of the *pdg* is generated by introducing a new symbol for each definition of a variable in the *pdg*. Cytron et al. [31] suggest a method to generate the minimal SSA form for a given program. When many definitions for a variable reach a particular node, a ϕ -function is introduced for that variable at that node. The ϕ -function represents a special type of function that takes a variable number of arguments as input and outputs a single value. The net effect of introducing a ϕ -function at a merge node is that only a single incoming value will pass through. The variable for which the ϕ -function is introduced, is assigned the return value of the function. Since renaming every new assignment is not very practical because of the obvious limitations in the size of the symbol table, most implementations provide def-use [1] links for each of the new definitions.

Special ϕ -functions are inserted at points where more than one definition of a variable reaches a node. If more than one control flow edge is incident on a node, there is a possibility that more than one definition of any variable reaches that node. The placing of ϕ -functions is a non-trivial problem because minimal number of them are to be generated for each of the program variables. Even though placing unnecessary ϕ -functions may generate a correct SSA form, it adds overhead to the optimization or transformation process for which the SSA form is used. The minimal SSA form can be generated by calculating the dominator [1] information. Consider two nodes b_i and b_j in a program dependence graph. At the node b_i a variable a is defined and is used in some computation in b_j . If the only path to node b_j from *ENTRY* has to pass through b_i , then b_i is said to strictly dominate b_j . When b_i strictly dominates b_j , the value of the variable a that reaches b_j has to come from b_i . Hence, in this case a ϕ -function for a is not needed before entrance to b_j . For instance, if there were other paths from *ENTRY* to node b_j , and the path through b_i was just one of the many, then b_j would be in the *dominance frontier* of b_i . In this case, not only does the definition of a in b_i reach b_j but also other definitions of a reach b_j . Hence, a ϕ -function for a just before the entrance to b_j is required. Calculating the dominance frontier information helps generate the minimal SSA form.

Figure 6.1 shows two versions of the same piece of code. Statements S1-S7 present the original version of the code. Statements T1-T8 depict the SSA form of the code. In statement T6, a ϕ -function has been placed because two definitions of the variable b were reaching statement S6 (code in original form). All variables have been renamed so that only a single assignment is made to each variable.

Various definitions reach the merge nodes (ϕ -function nodes) in the SSA form of the *pdg*. The information reaching the ϕ -function nodes are the different values. If the control flow information is also made to be an input to these ϕ -functions, then one can interpret which values will be assigned at these merge nodes. The problem is solved by using a *gated* single assignment form of the *pdg* [9]. The gated single assignment form replaces the original ϕ -functions with gating functions, which carry enough control information to interpret the

values at the merge nodes. There are three types of gating functions introduced. They are as follows:

γ : The γ -functions are introduced to capture the if-then-else condition. The ϕ -function, shown in statement T6 in Figure 6.1, would be replaced by a function such as $\gamma(a_1, b_2, b_1)$. Hence the statement T6 in Figure 6.1 would be replaced by the following statement:

$$\text{T6} \quad b_3 = \gamma(a_1, b_2, b_1)$$

When the value of a_1 is true, then b_3 will take on the value b_2 . The gated single assignment form gives far more information than the original ϕ -functions.

μ : The μ -functions are used to analyze the value flows inside a loop. These functions are generated for each of the variables defined inside the loop body. They are placed at the loop header and the function has three arguments: The first argument is a predicate that determines whether control will pass into the loop body; the second argument is the definition of the variable that is entering the loop before any iterations have been executed; the third argument is the definition of the variable that reaches the loop header after a complete iteration.

η : These function are placed at the loop exit, and they return the loop exiting definition of a variable. An η - function is placed at the loop exit for each of the definitions that flow out.

For this implementation, a variant of the gated single assignment form of the *pdg* called the *thinned gated* single assignment (TGSA) form has been used. The TGSA form of the *pdg* was developed at Rice University [48, 49, 47] and is part of the Parascope [29] environment.

6.2 Program Slicing

Program slicing is a source to source transformation technique suggested by Weiser [103]. The transformation finds every statement in a program that affects the value of any variable

```
S1  Input a,b
S2  if (a) then
S3      a = a - b
S4      b = b - a
S5      if (a) then
S6          a = a + 5
          endif
      endif
S7  Output a
S8  Output b
```

Figure 6.2: Code for Program Slicing

at any *point* in the program. A point in the program may be defined as an expression in any statement in the program body. The program slice that is generated for some expression \mathcal{E} in the program, when executed, should evaluate expression (\mathcal{E}) values identical to that of the original program for all inputs. Program slicing can be effectively used for analysis, debugging, testing of programs, parallelization and automatic integration of program version. A program slice is defined with respect to a statement \mathcal{S} in program \mathcal{P} and an expression \mathcal{E} in \mathcal{S} , as the statements and predicates of \mathcal{S} that might effect the value of \mathcal{E} in \mathcal{S} . Slicing criteria of a program \mathcal{P} is a tuple $\langle \mathcal{S}, \mathcal{E} \rangle$, where \mathcal{S} is a statement of the program \mathcal{P} , and \mathcal{E} is an expression in the statement \mathcal{S} .

Programs can be thought of as multiple threads, each of which computes a particular variable. These threads may or may not overlap one another. During program slicing, the thread for the variable based on which the slicing is being performed is found. Construction of program slices is complicated by nested structure. For a straight line code with no intricate control structures, one has to follow the use-def chains to get a complete slice. Since most programs have many control structures, a sophisticated version of the use-def

```

S1  Input a,b
S2  if (a) then
S3    a = a - b
S5    if (a) then
S6      a = a + 5
      endif
    endif
  endif

```

Figure 6.3: Slice for Slicing Criteria : $\langle S7, a \rangle$

chaining mechanism is required.

For a given program, a number of slices based on different slicing criteria can be generated. There always exists at least one slice for a given program and a slicing criteria, i.e., the program itself. It is desired that a slice of a program for a given slicing criteria be *statement minimal*. For a given program, \mathcal{P} and a given slicing criteria \mathcal{C} , the generated slice S is said to be statement minimal, if no other slice for \mathcal{C} on \mathcal{P} can be generated with a lesser number of statements. Proving that a slice is a statement minimal slice is undecidable. Weiser in his informal proof reduces the halting problem to that of finding a statement-minimal slice [103].

Figure 6.2 depicts the program to be used for program slicing. There are two variables in the program namely, a and b . Two different slicing criteria will be used to generate the slices. Figure 6.3 depicts the slice generated when the slicing is performed, based on statement $S7$ and variable a . The slice shown in Figure 6.3 has been generated by removing statements $S4$, $S7$ and $S8$ from the original code. The removed statements do not have any bearing on the value of the expression based on which the slicing is performed. Statements $S1$, $S3$ and $S6$ are introduced into the slice because variable a is being assigned a new value in these statements. Statements $S2$ and $S5$ are introduced into the slice because of control dependence. The conditional in statement $S2$ controls the assignment to variable

```

S1  Input a,b
S2  if (a) then
S4    b = b - a
      endif

```

Figure 6.4: Slice for Slicing Criteria : $\langle S8, b \rangle$

a in statements S3 and S6. The conditional in statement S5 controls the assignment to variable a in statement S6.

Figure 6.4 depicts another program slice in which slicing is done based on statement S8 and the variable b. Note that the slice has substantially fewer statements compared to the slice shown in Figure 6.3. Figure 6.2 shows that variable b has been used in statements S3 and S4. In statement S3, variable b is used but not defined, hence it is not included in the slice. On the other hand, statement S4 is where b is being defined, hence it is included in the slice. Statement S4 is executed if the value of the conditional in statement S2 is computed to be true. Statement S2 is introduced into the slice because of control dependence; statement S1 is present in the slice because it reads in the value of the variables a and b.

6.3 Program Slice Generation

From the algorithms described in Chapter 5, it is known that the generation of slices is a very important part of the transformation process. For every node in the slice graph, a slice must be generated. Popular dataflow algorithms [103] can be used to generate slices but the process is very time consuming. Generating slices efficiently can be done by using the *pdg* [55].

A *pdg*, for the example program shown in Figure 6.2, is depicted in Figure 6.5. Each node in the *pdg* represents a statement in the program. The nodes are marked by the statement numbers. All data dependencies in the program are shown by the solid arrows, while the

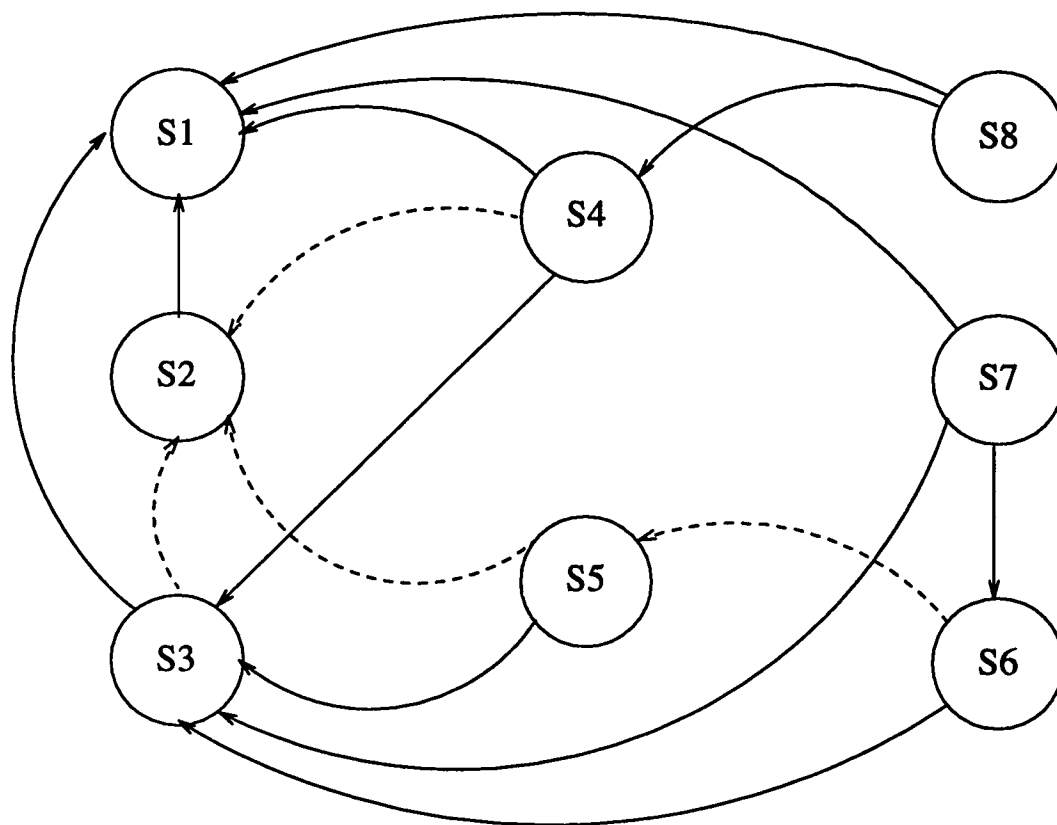


Figure 6.5: Program Dependence Graph for Slicing Example

control dependencies are shown by the dotted arrows. Using the *pdg* the slice shown in Figure 6.3 can be easily generated. Starting from node *S7* (Slicing criteria : $\langle S7, a \rangle$), all the reaching definitions of the variable *a* are found and they are nodes *S6*, *S3* and *S1*. Starting from each of the new nodes, all the nodes that are reachable are gathered. The complete set of nodes provides the slice. Starting from *S6*, nodes *S5* and *S3* can be reached. Again starting from node *S3*, *S2* and *S1* are reached. At this point, all nodes reachable from *S6*, *S3* and *S1* have been collected, and they are the nodes *S1*, *S2*, *S3*, *S5* and *S6*. The different nodes represents the slice.

Similarly to generate the slice shown in Figure 6.4, starting from node *S8* (Slicing criteria : $\langle S8, b \rangle$) all the reaching definitions of the variable *b* are found. Starting from node *S8*, nodes *S4*, *S2* and *S1* can be reached. Hence the slice is given by nodes *S1*, *S2* and *S4*. It follows that slice generation becomes a problem of simple graph traversal.

The program slicer that has been developed takes as input an abstract syntax tree (AST) node and a *pdg*. The AST node is equivalent to the slicing criteria. The AST node is mapped onto its corresponding node in the *pdg*. Starting from this node, all reachable nodes are found. Each new node encountered becomes a part of the slice. Use-def (ud) chaining [1] must be performed to find all of the reaching definitions. In conjunction to ud-chaining, the control dependence paths are followed to get a complete slice. The slice must be generated so that it can be placed (the target node) at the beginning of the procedure without changing the meaning of the program. Generation of a slice where the target node is at the beginning of the procedure fails, when there is a statement *S* which modifies a distributed array and the statement also happens to be part of the slice being generated. In this case, the node *T* in the graph whose dominator is the node corresponding to *S* is found and made the target node. Having this constraint imposed on the generation of the target node allows working with any type of irregular code.

6.4 Transformation Implementation

This section includes a high level description of how the transformation discussed in the previous chapter has been implemented. The actual transformation uses the different tools that were described in this chapter. The transformations have been implemented in the Parascope environment. The regular part of the Fortran D compiler analyzes the input code, collects the array references it cannot handle and calls the irregular part of the compiler.

The set of irregular references are passed to the slice graph generation procedure. Each unique slice and its target node become a node in the slice graph. After the nodes in the slice graph have been built, the edges of the graph are generated using the dependencies that exist between the slices. A topological sorting routine is called with the generated slice graph. After the sorting, inspector/executor pairs are created for each of the nodes in the slice graph.

Various loops with more than one level of indirection have been run through the transformation process. Progress is being made to further generalize this method and generate a more robust implementation.

Chapter 7

Conclusion and Future Work

The work presented here explains in detail the type of compiler support and transformation techniques required to parallelize irregular programs written in data parallel languages. The information provided here will be invaluable to anyone writing a parallelizing compiler for irregular problems. This chapter summarizes the contributions made by this dissertation followed by the direction of future research.

7.1 Contributions of this Thesis

There has been some preliminary work in the area of automatic parallelization of irregular problems [60]. Tools have been developed to generate inspector/executor pairs, but they lack the optimizations necessary to parallelize any real application codes. The contributions of this thesis have been in two different but related areas. They are:

- A compiler runtime compilation system has been designed and developed to help parallelize irregular loops.
- Transformation techniques have been suggested that allow for automatic parallelization of real irregular applications.

7.1.1 Development of Compiler Support

An efficient runtime compilation system has been designed and implemented. The compilation system is comprised of a set of highly optimized tools that can be used to automatically generate inspector/executor pairs for irregular loops. A variety of different irregular application codes were studied and, based on experience with these codes, tools were developed.

The development of software caching methods is an important contribution of this work. Techniques have been developed for caching off-processor data. *Incremental scheduling* is an important concept that has been developed to optimize off-processor data caching. There are a number of application codes (especially particle codes) that would be nearly impossible to parallelize on the available distributed memory hardware without using the basic ideas of incremental scheduling. Compiler transformations designed to use incremental scheduling automatically were developed as part of a joint project with a group at Rice University [46] but have not been presented in this dissertation.

A highly scalable global to local address translation mechanism has been developed. This addressing uses a *paged distributed translation table*, which stores all required information. This mechanism will be useful when parallelizing highly adaptive irregular application codes. This thesis presented a detailed performance analysis of the various tools, using both a synthetic workload generator and a number of actual application codes. The parallelization of the actual application codes was done to show the efficiency of the methods developed here.

7.1.2 Compiler Transformation

This dissertation presented a method to automatically parallelize irregular applications for execution on distributed memory machines. This operation is accomplished by transforming irregular loops into inspector/executor pairs. The generation of inspector/executor pairs for loops with a single level of indirection has been accomplished by both this author and other researchers [60, 18]. But most irregular loops found in application codes have indirection patterns that are not easily deciphered. Hence the original code must be transformed

into an intermediate state so that the inspector/executor transformation can be applied to parallelize it.

This thesis presented algorithms that can be used to parallelize irregular codes with multiple levels of indirection. The method is based on program slicing techniques. The algorithms presented are very general and work for all irregular codes. The original code is transformed until there is, at most, a single level of indirection. The single level of indirection is achieved by peeling off each level of indirection until loops in the code have only a single level of indirection. At this point, the inspector/executor transformation is applied.

7.2 Future Work

This thesis has been one of the more serious efforts to automatically parallelize real irregular applications codes. Techniques have been developed for a subset of irregular problems; the loosely synchronous variety comprises 25% to 30% of the irregular applications. There are, however, irregular applications which cannot be effectively parallelized by the techniques presented here.

Some irregular codes are highly adaptive; the indirection arrays change every iteration. In such cases, the inspector/executor type of parallelization is not effective because the cost of generating the inspectors cannot be amortized. If the inspector/executor type of computation is used to parallelize such applications, a large percentage of the time will be invested in inspector generation. Overlapping communication and computation in such highly adaptive codes may be very useful. One might generate partial inspectors; start a phase of data communication and, while the data is being moved, generate the rest of the inspector. This procedure would require some form of loop stripmining.

The possibility of using interprocedural slicing [55] to generate inspectors should be explored. Such an approach might allow an experimenter to generate highly efficient parallel code.

This author would like to extend the methods developed in this thesis to handle applications that have distinct phases of computations, where each of the phases can be either regular or irregular (Example: particle-in-cell codes [39, 102]). Such computations require additional tools to handle the regular parts of the codes and also the extension of irregular tools to do efficient data movements. The data partitioning between the different phases must be performed efficiently to obtain effective parallelization. Development of compiler transformations to automatically parallelize such codes is indeed challenging.

The area of automatic parallelization of irregular codes is very new in the parallel compiler world. A great deal of work remains; this thesis has provided a solid foundation for exploring these issues.

Bibliography

- [1] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.
- [2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, June 1987.
- [3] J. R. Allen and K. Kennedy. PFC: A program to convert Fortran to parallel form. In K. Hwang, editor, *Supercomputers: Design and Applications*, pages 186–203. IEEE Computer Society Press, Silver Spring, MD, 1984.
- [4] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.
- [5] Randy Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.
- [6] F. André, J. Pazat, and H. Thomas. Data distribution in Pandore. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.
- [7] F. André, J. Pazat, and H. Thomas. Pandore: A system to manage data distribution. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.
- [8] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J.A. Webb. The warp computer: Architecture, implementation, and performance. *IEEE Trans. on Computers*, C-36(12):1523–1538, December 1987.
- [9] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271, June 1990.
- [10] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.
- [11] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

- [12] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 168–175, 1990.
- [13] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for scientific programs on distributed memory machines. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, December 1989.
- [14] Harry Berryman, Joel Saltz, and Jeffrey Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.
- [15] S. Bokhari. Communication overhead on the intel ipsc-860 hypercube. Report 90-10, ICASE Interim Report, 1990.
- [16] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, Sanjay Ranka, and Min-You Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. To appear in the *Journal of Parallel and Distributed Computing*, March 1993.
- [17] Zeki Bozkus, Sanjay Ranka, and Geoffrey Fox. Benchmarking the cm-5 multicomputer. To appear in *Frontiers '92*, 1992.
- [18] P. Brezany, M. Gerndt, V. Sipkova, and H.P. Zima. SUPERB support for irregular scientific computations. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-92)*, pages 314–321. IEEE Computer Society Press, April 1992.
- [19] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187, 1983.
- [20] B. R. Brooks and M. Hodoscek. Parallelization of charmm for mimd machines. *Chemical Design Automation News*, 7:16, 1992.
- [21] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.
- [22] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.
- [23] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. Technical Report 92-9, ICASE, NASA Langley Research Center, March 1992.
- [24] C. Chase, A. Cheung, A. Reeves, and M. Smith. Paragon: A parallel programming environment for scientific applications using communication structures. In *Proceedings of the 1991 International Conference on Parallel Processing*, St. Charles, IL, August 1991.
- [25] M. Chen, Y. Choo, and J. Li. Theory and pragmatics of compiling efficient parallel code. Technical Report YALEU/DCS/TR-760, Dept. of Computer Science, Yale University, New Haven, CT, December 1989.

- [26] M. C. Chen. A parallel language and its compilation to multiprocessor architectures or VLSI. In *2nd ACM Symposium on Principles of Programming Languages*, January 1986.
- [27] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and J. Saltz. Software support for irregular and loosely synchronous problems. *Computing Systems in Engineering*, 3(1-4):43-52, 1992. Papers presented at the Symposium on High-Performance Computing for Flight Vehicles, December 1992.
- [28] The connection machine cm-5 technical summary. Report, Thinking Machines Corporation, 1991.
- [29] Keith D. Cooper, Mary W. Hall, Robert T. Hood, Ken Kennedy, Kathryn S. McKinley, John M. Mellor-Crummey, Linda Torczon, and Scott K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244-263, February 1993. In Special Section on Languages and Compilers for Parallel Machines.
- [30] Thinking Machines Corporation. CM Fortran reference manual. Technical Report version 1.0, Thinking Machines Corporation, Feb 1991.
- [31] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the Sixteenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1989.
- [32] W. J. Dally, J. A. Stuart Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, and P. R. Nuth. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEEEM*, pages 23-39, April 1992.
- [33] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives. In *To appear AIAA Journal, AIAA-92-0562*, Jan 1992.
- [34] R. Das and J. Saltz. Parallelizing molecular dynamics codes using parti software primitives. In *Parallel Processing for Scientific Computation, Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, Norfolk VA, March 1993*, 1993.
- [35] R. Das, J. Saltz, D. Mavriplis, and R. Ponnusamy. The incremental scheduler. In *Unstructured Scientific Computation on Scalable Multiprocessors*, Cambridge Mass, 1992. MIT Press.
- [36] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. Technical Report CS-TR-3076 and UMIACS-TR-93-42, University of Maryland, Department of Computer Science and UMIACS, May 1993. Appears in LCPC '93.
- [37] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. Technical Report CS-TR-3163 and UMIACS-TR-93-109, University of Maryland, Department of Computer Science and UMIACS, October 1993. Submitted to Journal of Parallel and Distributed Computing.

- [38] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [39] R. D. Ferraro, P. C. Liewer, and V. K. Decyk. Dynamic load balancing for a 2d concurrent plasma pic code. *submitted to Journal of Computational Physics*, 1991.
- [40] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Department of Computer Science Rice COMP TR90-141, Rice University, December 1990.
- [41] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. Fortran D specification. Technical Report, Dept. of Computer Science, Rice University, 1991.
- [42] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Computers*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
- [43] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.
- [44] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, December 1989.
- [45] M. W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. Technical Report TR91-169, Dept. of Computer Science, Rice University, November 1991.
- [46] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.
- [47] Paul Havlak. Personal communications.
- [48] Paul Havlak. Construction of thinned gated single-assignment form. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.
- [49] Paul Havlak. *Interprocedural Symbolic Analysis, in preperation*. PhD thesis, Rice University, Houston, TX, 1993.
- [50] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.
- [51] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the Sixth International Conference on Supercomputing*. ACM Press, July 1992.
- [52] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12:415–422, August 1991.

- [53] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings Supercomputing '91*, pages 86–100. IEEE Computer Society Press, November 1991.
- [54] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.
- [55] S. Horowitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the SIGPLAN '88 Conference on Program Language Design and Implementation*, Atlanta, GA, June 1988.
- [56] High performance fortran language specification : Version 1.0. Report, High Performance Fortran Forum, 1993.
- [57] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.
- [58] A. Jameson, T. J. Baker, and N. P. Weatherhill. Calculation of inviscid transonic flow over a complete aircraft. *AIAA paper 86-0103*, January 1986.
- [59] A. K. Jones, R. J. Chansler, I. Duram, K. Schwans, and S. R. Vegdahl. Staros, a multiprocessor operating system for the support of task forces. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 117–127, 1979.
- [60] C. Koelbel. *Compiling Programs for Nonshared Memory Machines*. PhD thesis, Purdue University, West Lafayette, IN, August 1990.
- [61] C. Koelbel. Compile-time generation of regular communications patterns. In *Proceedings of Supercomputing '91*, pages 101–110, Albuquerque, NM, November 1991.
- [62] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.
- [63] C. Koelbel and P. Mehrotra. Programming data parallel algorithms on distributed memory machines using Kali. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.
- [64] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOP)*, pages 177–186. ACM Press, March 1990.
- [65] D. Kuck, R. Kuhn, B. Leasure, and M. J. Wolfe. The structure of an advanced retargetable vectorizer. In *Proceedings of COMPSAC 80, the 4th International Computer Software and Applications Conference*, pages 709–715, Chicago, IL, October 1980.
- [66] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

- [67] Monica Lam. *A Systolic Array Optimizing Compiler*. PhD thesis, Carnegie Mellon University, May 1987. Also available as Technical Report CMU-CS-87-187.
- [68] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159. IEEE Computer Society Press, May 1990.
- [69] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.
- [70] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proceedings of Supercomputing '90*, New York, NY, November 1990.
- [71] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.
- [72] J. Li and M. Chen. Synthesis of explicit communication from shared-memory program references. Technical Report YALEU/DCS/TR-755, Dept. of Computer Science, Yale University, New Haven, CT, May 1990.
- [73] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239. ACM Press, 1986.
- [74] R. Littlefield. Efficient iteration in data-parallel programs with irregular and dynamically distributed data structures. Technical Report 90-02-06, University of Washington, Dept. of Computer Science and Engineering, Seattle, WA, February 1990.
- [75] D. J. Mavriplis. Three dimensional multigrid for the Euler equations. *AIAA paper 91-1549CP*, pages 824–831, June 1991.
- [76] James McGraw, Stephen Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas. *Sisal: Streams and Iteration in a Single Assignment Language, Language Reference Manual Version 1.2*. Lawrence Livermore National Laboratory, March 1985.
- [77] P. Mehrotra and J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.
- [78] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, July 1988.
- [79] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu. Medusa: An experiment in distributed operating system structures. *CACM*, Feb 1980.
- [80] W. Appelbe P. Dasgupta, R. Leblanc. The Clouds distributed operating system: Functional description, implementation details and related work. In *IEEE International Conference on Distributed Computing Systems*, 1988.

- [81] Parasoftware Corporation. *Express User's Manual*, 1989.
- [82] Paul Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Vol. 1*, pages 384–390. ACM Press, January 1988.
- [83] Michael J. Quinn and Philip J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, 7(5):69–76, September 1990.
- [84] A. Rogers. *Compiling for Locality of Reference*. PhD thesis, Cornell University, Ithaca, NY, June 1990.
- [85] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.
- [86] J. Rose and G. Steele, Jr. C*: An extended C language for data parallel programming. Technical Report PL87-5, Thinking Machines, Inc, 1986.
- [87] M. Rosing. *Efficient Language Constructs for Complex Data Parallelism on Distributed Memory Multiprocessors*. PhD thesis, Dept. of Computer Science, University of Colorado, November 1991.
- [88] M. Rosing and R. Schnabel. An overview of Dino - a new language for numerical computation on distributed memory multiprocessors. Technical Report CU-CS-385-88, University of Colorado, Boulder, 1988.
- [89] M. Rosing, R. Schnabel, and R. Weaver. Expressing complex parallel algorithms in DINO. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.
- [90] M. Rosing, R. Schnabel, and R. Weaver. Massive parallelism and process contraction in Dino. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, December 1989.
- [91] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. Technical Report CU-CS-457-90, Dept. of Computer Science, University of Colorado, April 1990.
- [92] Y. Saad. Sparsekit: a basic tool kit for sparse matrix computations. Report 90-20, RIACS, 1990.
- [93] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.
- [94] J. Saltz, K. Crowley, R. Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.
- [95] A. Sussman. Model-driven mapping onto distributed memory parallel computers. In *Proceedings Supercomputing '92*, pages 818–829. IEEE Computer Society Press, November 1992.

- [96] Alan Sussman. *Model-Driven Mapping of Computation onto Distributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, September 1991. Also available as Technical Report CMU-CS-91-187.
- [97] P.-S. Tseng. A systolic array parallelizing compiler. *Journal of Parallel and Distributed Computing*, 9(2):116–127, June 1990.
- [98] Ping-Sheng Tseng. *A Parallelizing Compiler For Distributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, May 1989. Also available as Technical Report CMU-CS-89-148.
- [99] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 107–119, June 1991.
- [100] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266. ACM Press, May 1992.
- [101] Reinhard von Hanxleden, Ken Kennedy, Charles Koelbel, Raja Das, and Joel Saltz. Compiler analysis for irregular problems in Fortran D. Technical Report 92-22, ICASE, NASA Langley Research Center, June 1992.
- [102] D. W. Walker. Characterizing the parallel performance of a large-scale, particle-in-cell plasma simulation code. *Concurrency Practice and Experience*, 4:257, 1990.
- [103] M. Weiser. Program slicing. *IEEE Trans. on Software Eng.*, SE-10(4):352–357, July 1984.
- [104] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 2, pages 26–30, 1991.
- [105] W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *CACM*, June 1974.
- [106] H. Zima, H.-J. Bast, and M. Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.
- [107] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran — a language specification, version 1.1. Interim Report 21, ICASE, NASA Langley Research Center, March 1992.

VITA

Subhendu Das was born in Calcutta, India on September 23, 1961. After graduating from St. Xaviers Collegiate School in Calcutta, he attended Jadavpur University in his home state and received a B.S. degree in Mechanical Engineering, June 1984. He then continued his education at Clemson University in Clemson, South Carolina and received an M.S. degree in Mechanical Engineering, May 1987. In January 1989, Mr. Das entered the College of William and Mary as a graduate assistant in the Department of Computer Science.