Compiler Optimization for Parallel Computation

by

Wen-Shiu Chyr

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Master of Science
in
Computer Science

Winnipeg, Manitoba, Canada, 1995

ISBN 0-612-13034-7

Name Computer Science

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science
SUBJECT TERM

0984
SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

COMPILER OPTIMIZATION FOR PARALLEL COMPUTATION

BY

WEN-SHIU CHYR

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Abstract

The dependence structure of a parallel computation can be characterized by a directed acyclic graph (DAG) where nodes and edges represent tasks and dependences respectively. These edges define message-passing synchronization in a distributed-memory model. If a dependence edge in the DAG is transitive, then there is an alternate path composed of edges connecting source and sink node of that dependence. The startup cost of a message transmission is typically higher than actual data transfer. Communication overhead can be reduced by grouping messages. Message transfers along transitive edges can be avoided, without losing parallelism, by appending the data to the messages on the alternate path that covers the transitive edge. In this thesis, a compiler-optimization technique for pipelined computation is introduced. This algorithm: (1) identifies the dependences corresponding to transitive edges in simple loops with constant dependences; and (2) introduces a scheme for assigning storage for nonlocal data, corresponding to the transitive edges, in the node program.

Acknowledgements

I wish to express my sincere gratitude to Dr. Ken Barker who generously granted his time to proof-read my thesis and gave me many valuable suggestions. I deeply appreciate Dr. Ken's support in coordinating my thesis defence.

I also take this opportunity to thank both examiners, Dr. David Blight (Department of Electrical and Computer Engineering) and Dr. Kasi Periyasamy (Department of Computer Science), both of whom took time to read through my thesis.

I wish to express my heartfelt thanks to Dr. Prasad Krothapalli who inspired this thesis and essentially made it possible. Despite a busy schedule, Dr. Prasad always made sure to enquire about, criticize, and encourage my research. Thank you again.

Finally, I would like to thank my family for their never-ending love and support.

Contents

1 Introduction 1

1.1 Organization 3

2 Parallel Processors and Parallel Programming 4

2.1 The Evolution of Computer Architecture 7

2.1.1 Flynn's Taxonomy 9

2.1.2 MIMD Parallel Processors 11

2.1.3 Distributed-Memory Multicomputers 12

2.2 Parallel Programming 14

2.3 Related Work 18

2.4 Summary 20

3 FORTRAN D 21

3.1 FORTRAN D Language 22

3.1.1 Problem Mapping 23

3.1.2 Machine Mapping 28

3.1.3 Additional Features of FORTRAN D 31

3.2 Summary 34

4 FORTRAN D Compiler 36

4.1 Program Analysis 37

4.1.1 Dependence Analysis 37

4.1.2 Data Decomposition Analysis 39
4.1.3 Partitioning Analysis 40
4.1.4 Communication Analysis 47
4.2 Program Optimization 47
4.3 Code Generation 51
4.4 Summary 57

5 FORTRAN D Compiler Optimizations 58
5.1 Reducing Communication Overhead 59
5.1.1 Message Vectorization 60
5.1.2 Message Coalescing 60
5.1.3 Message Aggregation 61
5.1.4 Collective Communication 64
5.2 Hiding Communication Overhead 64
5.2.1 Message Pipelining 64
5.2.2 Vector Message Pipelining 67
5.2.3 Iteration Reordering 70
5.2.4 Nonblocking Message 73
5.3 Exploiting Parallelism 74
5.3.1 Partitioning Computation 74
5.3.2 Reductions and Scans 74
5.3.3 Dynamic Data Decomposition 78
5.3.4 Pipelining Computation 78
5.4 Reducing Storage 83
5.5 Optimization Algorithm 83
5.6 Program Transformations 85
5.7 Summary 88

6 Compiler Optimization for Parallel Computation **89**

6.1 Iteration Space Dependence Graph 91

6.2 Identifying Transitive-Edge Dependences 92

6.2.1 Regular Non-Nested Loops 95

6.2.2 Regular Double-Nested Loops 99

6.3 Implementation on the FORTRAN D Compiler 104

6.3.1 Modifying Communication Analysis 106

6.4 Program Transformation 107

6.5 Analytical Evaluation 114

6.6 Conclusions 115

7 Conclusions **116**

7.1 Future Work 118

List of Figures

2.1 Grand Challenge Requirements (quoted from [1], pg. 121) 6

2.2 .Computer Architecture Evolution from Sequential Scalar Computers to Vector Processors and Parallel Computers (quoted from [1], pg. 10) 8

2.3 Flynn's Classification of Computer Architectures (quoted from [1], pg. 12) 10

2.4 Shared-Memory Multiprocessors (quoted from [1], pg. 20-25) 13

2.5 Two Approaches of Parallel Programming (quoted from [1], pg. 19) 16

3.1 Example of Problem Mapping 23

3.2 Example of Problem Mapping Statements 25

3.3 Example of Array Permutation, Collapse, and Embedding 27

3.4 Example of Alignment Options 28

3.5 Example of Data Distribution 30

3.6 Example of Processor Distribution 31

3.7 Example of Program Segment with Irregular Distribution 32

3.8 Example of Irregular Distribution 32

3.9 Example of FORALL Loop 33

4.1 Sequential Loop Nest 39

4.2 Example of RSD Representation 42

4.3 Example of Global and Local Indices 42

4.4 Example of a Fortran D Program(Jacobi) 43

4.5 The Augmented Iteration Sets of Jacobi Program 46

4.6 Example of Scalar Variables 46
4.7 Example of Index Sets and Nonlocal Index Sets for S1 in Jacobi Program 48
4.8 Example of Run-time Resolution and Message Vectorization 49
4.9 The SPMD Code of Jacobi Program 52
4.10 Example of Message Generation for Loop-Carried and Loop-Independent Dependences 54

5.1 Example of Message Vectorization and Message Coalescing (Livermore 7-Equation of State Fragment)[14] 62
5.2 Example of Message Vectorization and Coalescing (Compiler Output) 63
5.3 Example of Message Aggregation (Livermore 18-Explicit Hydrodynamics)[14] 65
5.4 Example of Message Aggregation (Compiler Output) 66
5.5 Example of Vector Message Pipelining (Red-Black SOR) [14] 68
5.6 Example of Vector Message Pipelining (Compiler Output) 69
5.7 Example of Iteration Reordering 71
5.8 Example of Iteration Reordering (Compiler Output) 72
5.9 Example of Reduction (Inner Product) [14] 76
5.10 Example of Scan (First Sum) 77
5.11 Example of Dynamic Data Decomposition (ADI Integration) [13] . . 79
5.12 Parallel and Pipelined Computation 80
5.13 Example of Pipelining Computation (Implicit Hydrodynamics) . . . 81
5.14 Example of Pipelining Computation (Implicit Hydrodynamics) . . . 82
5.15 Compiler Optimization Algorithm 84
5.16 Loop Distribution 86
5.17 Loop Fusion 86
5.18 Loop Interchange 87
5.19 Strip Mining 88

6.1 Eliminating Messages for Pipelined Computation with Transitive-Edge Dependences 90

6.2 Example of ISDG 93
6.3 SPMD Program 94
6.4 Example of R_subgraph and Transitive-Edge Dependences 96
6.5 Sequential Execution Order Depth First Search Algorithm 97
6.6 Sequential Execution Order Depth First Search Algorithm(Cont.) . 98
6.7 Case One Illustrated 101
6.8 Case Two Illustrated 103
6.9 Case Three Illustrated 105
6.10 Source code transformation for A(i-d_i) 107
6.11 Example of Source Code Transformation for Nonnested Loop 108
6.12 Compiler Output 109
6.13 Example of Fortran D Program Segment 110
6.14 Source Code Transformation for A(i-di1,j-di2) 111
6.15 Code Transformation for the Program in Case One 112
6.16 Code Transformation for the Program in Case Two 113
6.17 Code Transformation for the Program in Case Three 114

Chapter 1

Introduction

Conventional parallel computers are classified, based on their architectures, into two groups: shared memory multiprocessors, and distributed memory multicomputers. Each makes different tradeoffs with respect to scalability and programmability but future parallel architectures are likely to reflect aspects of these two models. Developing high-level parallel programming languages that are easy to program and portable across multiple architectures is a central issue at current research. Ideally, compilers that generate efficient parallel code must be developed so program efficiency does not rely on programmer expertise. Such a compiler would exploit the inherent parallelism in a program so the generated parallel code has a minimum at communication overhead. This thesis presents an optimization for a parallelizing compiler to reduce communication overhead on distributed-memory machines, which is the most difficult programming model among parallel architectures.

Eliminating of redundant dependences has been shown to reduce synchronization costs in a shared-memory multiprocessor [12]. A computation's dependence struc-

ture is captured by a directed acyclic graph (DAG), where nodes represent tasks and edges represent dependences between tasks. Krothapalli and Sadayappan [12] demonstrate that the constraint imposed by a dependence edge is enforced by an appropriate synchronization instruction.

Most early research [11] in parallelizing compilers focused on data dependence analysis, loop transformations, and reducing synchronization costs in a shared memory multiprocessor. Since the introduction of Intel's hypercube [1], there has been increasing interest in distributed-memory machines which coordinate activities by exchanging messages. Unfortunately, for small messages, the message initiation is typically two orders of magnitude higher than the data transmission costs. So several optimizations that increase the message length in executing data-parallel loops have been proposed [14]. These optimizations increase the message length by combining several messages between two processors into one.

The dependency structure captured by the DAG used by Krothapalli and Sadayappan [12] is sufficient for shared memory architectures because the interprocessor communication overhead is effectively negligible. This is not true for the distributed-memory architecture. Krothapalli and Sadayappan also argue that the transitive edges are not required in their analysis, so they are effectively ignored, but this is because their primary concern is ensuring that all constraints are captured. This thesis argues that by exploiting the transitive edges in the DAG, message traffic can be reduced because small messages are combined to save communication overhead. The technique requires messages be rerouted based on the transitive edges found in the graph.

Discussions of optimizing parallel compilation requires a suitable compiler. The

compiler must support code parallelization, readily provide source code, and be extensible. The FORTRAN D compiler under development at Rice University provides a programming language and environment that meets these needs [13]. Although this language is freely available it is not currently well known. Therefore, this thesis carefully describes the language with particular attention paid to the parallel aspects. Further, the source code is available, so it is possible to extend the compiler and its data structures in interesting new ways. Therefore, this thesis also describes the FORTRAN D compiler and the corresponding optimizer. The description of the language, compiler, and optimizer provides a pragmatic way to discuss the central issues in parallelizing compilers.

1.1 Organization

Chapter 2 reviews parallel processors and parallel programming. FORTRAN D and the FORTRAN D compiler is described in Chapters 3 and 4, respectively. The FORTRAN D Compiler optimizations are discussed in Chapter 5. Chapter 6 presents important result from this thesis by proposing a new optimization technique as outlined above. Finally, Chapter 7 summarizes the problems, results and contributions at this thesis in addition to providing suggestions for further research directions.

Chapter 2

Parallel Processors and Parallel Programming

In 1992 the U.S. High-Performance Computing and Communication (HPCC) program announced the 'Grand Challenges' facing scientific computing (these are summarized in Table 2.1). The computing requirements for each challenge are shown in Figure 2.1. A computer system sufficient to meet all of these requirements will have one Teraflop of computing power, one Terabyte of main memory, in addition to one Terabyte per second of I/O bandwidth [1]. Unfortunately. This is "the goal of 3T performance". Unfortunately, , the most powerful conventional computers are still many orders of magnitude away from 3T performance. Therefore, parallel processing seems to be the only alternative that satisfies these requirements.

The balance of this chapter provides the fundamentals of parallelism in terms of both parallel processors and programming. Section 2.1 provides a historical perspective by describing the evolution of computer architectures over the past 20

Magnetic recording Technology	To study magnetostatic and exchange interactions to reduce noise in high density disks
Rational drug design	To develop drug to cure cancer or AIDS by blocking the action of HIV protease.
High-speed civil transport	To develop supersonic jets through computational fluid dynamics running on supercomputers.
Catalysis	Computer modeling of biomimetic catalysts to analyze enzymatic reactions in manufacturing process.
Fuel combustion	Designing better engine models via chemical kinetics calculations to reveal fluid mechanical effects.
Ocean modeling	Large-scale simulation of ocean activities and heat exchange with atmospherical flows.
Ozone depletion	To study chemical and dynamical mechanisms controlling the ozone depletion process
Digital anatomy	Real-time, clinical imaging, computed tomography, magnetic resonance imaging with computers.
Air pollution	Simulated air quality models running on supercomputers to provide more understanding of atmospheric systems.
Technology linking research to education	Scientic or engineering education aided by computer simulation in hetergeneous network systems.
Protein structure design	3-D structrual study of protein formation by using MPP system to perform computational simulations.
Image understanding	Use large supercomputers for producing the rendered images or animations in real time.

Table 2.1: Grand Challenge Applications (quoted from [1], pg. 119)

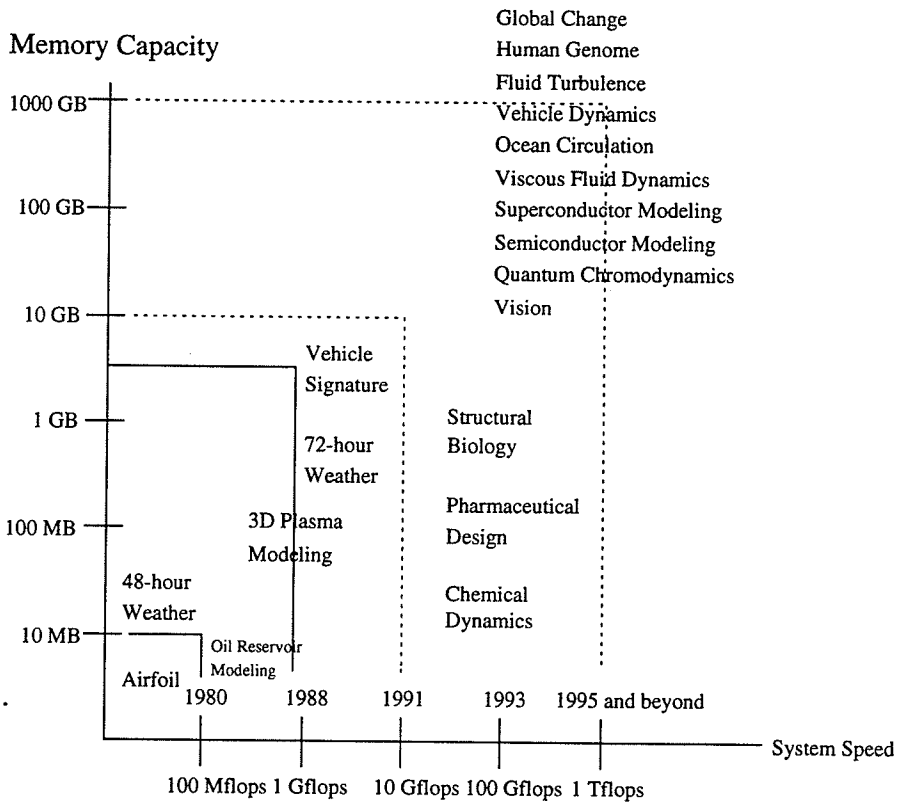


Figure 2.1: Grand Challenge Requirements (quoted from [1], pg. 121)

years. Particular attention is paid to Flynn's well known taxonomy though other classification schemes may work equally well. Secondly, section 2.2 introduces the fundamentals of parallel programming and provides a basis for the claim that FORTRAN D is an appropriate basis for the results presented in this basis.

2.1 The Evolution of Computer Architecture

Over the past two decades, demand for higher performance, lower cost, and sustained productivity in scientific applications has resulted in improvements in system performance. Major improvements achieved in technology, system architecture, system software, and system organization contributed to this increase in performance. Over the last two decades, technological advances are the major contributors to the increase in system speed. Due to the limitations imposed by laws of physics, the von Neumann architectures and the parallel computing seems to be the best alternative to improve system performance [1, 2, 3].

The evolution of computer architecture is illustrated in Figure 2.2 [1]. The scalar computer is the origin and it is based on von Neumann architecture, built as a sequential machine, for executing scalar data. The subsequent architectures adopted the *lookahead* approach which overlaps instruction prefetch/decode phase with execution phase. This technology led to *functional parallelism* that adopts *instruction pipelining* to allow multiple functional units to work concurrently. *Pipelining* decomposes a computation into a number of steps that can be assigned to processors in a sequence so multiple data streams can appear to be executed in "parallel". It can be applied to instruction execution, arithmetic computations, and memory-

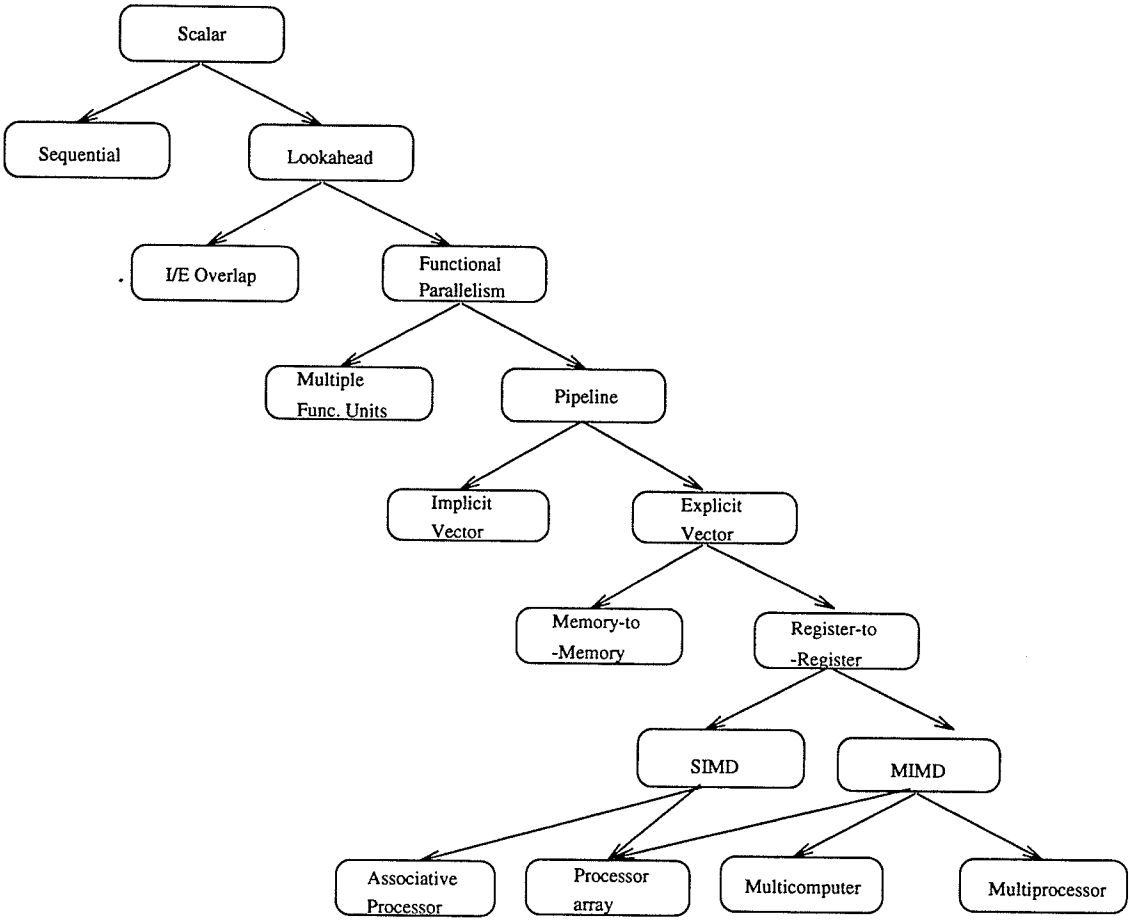


Figure 2,2: Computer Architecture Evolution from Sequential Scalar Computers to Vector Processors and Parallel Computers (quoted from [1], pg. 10)

access operations. The advanced vector computers are equipped with scalar and vector hardware. System performance of these machines is in order of hundreds of MFLOPS.

During the last decade, parallelism has become a well-known and popular strategy to achieve higher performance [2]; The advent of very large-scale integration (VLSI) technology is the principal contributor to parallel computer technology.

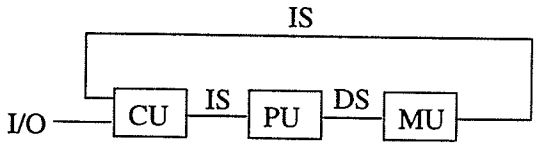
This spawned much research activity that developed both real and theoretical computers. Recently a number of commercial parallel computers, based on various architectural models, have appeared on the market in a wide range of prices [1].

2.1.1 Flynn's Taxonomy

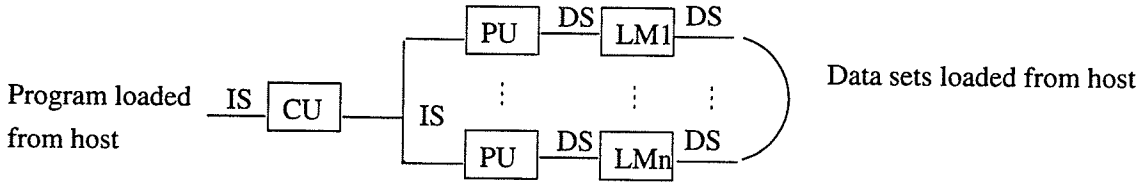
Flynn (1972) provided a useful taxonomy that helps us understand the principles of parallel architectures. It is based on the concepts of instruction and data streams. In this classification, the *multiplicity* of hardware is used to classify each architecture. "The multiplicity is defined as the maximum possible number of simultaneous operations (instructions) or operands (data) being in the same phase of execution at the most constrained component of the organization" [3].

Uniprocessor based computers are classified as *Single Instruction Stream, Single Data Stream* (SISD) computers. As shown in Figure 2.3(a), a SISD computer has one control unit (CU) and one processing unit (PU). In any instruction cycle the control unit issues an instruction to the processing unit. PU reads operands and executes the instruction. A computer with functional parallelism or pipelining is also classified as SISD computer, because it decodes only a single instruction in a unit time [1, 4].

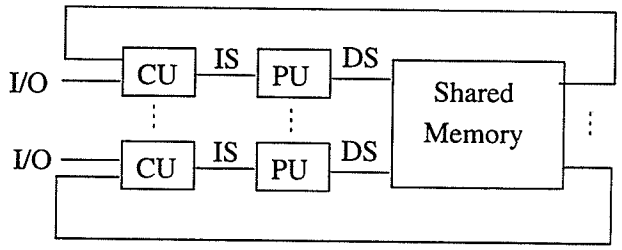
Single-Instruction Stream, Multiple Data Streams (SIMD) computers (Figure 2.3(b)) are computers with a single control unit (CU) to monitor a number of processing elements (PE). During an instruction cycle, the control unit broadcasts an instruction to all the PEs and all the PEs execute this instruction on different data. This



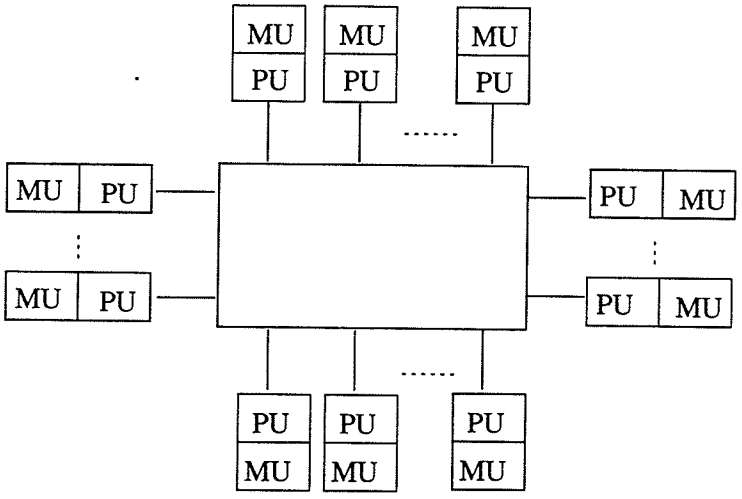
(a) SISD uniprocessor architecture



(b) SIMD uniprocessor architecture(with distributed memory)



(c) MIMD architecture with shared-memory



(d) MIMD architectures(with distributed-memory)

Figure 2.3: Flynn's Classification of Computer Architectures (quoted from [1], pg. 12)

model is also called *processor array* [4]. The SIMD machines such as Illiac IV and Thinking machine CM2 are not suitable for general purpose computations [1].

Most existing parallel computers fall into the class of *Multiple Instruction Streams, Multiple Data Streams* (MIMD). In a MIMD parallel computer, a separate control unit is associated with each processing unit. In an instruction cycle, each CU issues an instruction to its corresponding processing unit, so we have multiple instruction streams executed over multiple data streams.

2.1.2 MIMD Parallel Processors

The MIMD parallel computers can be further defined into two broad groups, namely, shared-memory multiprocessors and distributed-memory multicomputers (see Figure 2.3(c) and Figure 2.3(d), respectively). This section describes the former while the latter is discussed in section 2.1.3. In a shared-memory multiprocessor, synchronization between processors is achieved through shared variables. The following are most common shared-memory multiprocessor models:

The Uniform-Memory-Access (UMA) model: In this model, all the physical memory modules and peripherals are shared by all the processors. Figure 2.4(a) shows processors can access a shared global memory through a common bus, a crossbar switch, or a multi-stage interconnection network. The access time to each memory word in the global memory is the same for all the processors. The UMA model is the simplest processor intercommunication model and is suitable for general-purpose applications by multiple users. However, they

are not architecturally scalable [1]. Bus-based multiprocessors with twenty to thirty processors are commercially available.

The Nonuniform-Memory-Access (NUMA) model: In this model, the access time for each memory word depends on the location of the memory word. There are two types of memories in this machine model: local memory and remote memory. Each processor is directly connected to its local memory and all local memories together constitute the global shared-memory (Figure 2.4(b)). Local memory access time is an order of magnitude smaller than remote memory [1, 4].

The Cache-Only Memory Architecture (COMA) model: This model is similar to the NUMA model, except each local memory module is replaced with a cache. In a COMA machine, the binding between an address and a processor is dynamic. This model can be treated as cache-only NUMA machine. The shared memory is formed by all local caches in the system as shown in Figure 2.4(c). COMA machines have scalable architecture and research groups such as Stanford FLASH [5], I-ACOMA [6], and Data Diffusion Machines [7] are exploring this architecture. One benefit of this model is the simple programming model.

2.1.3 Distributed-Memory Multicomputers

The second MIMD machine is a distributed-memory multicomputer system where multiple computers exist but there is no shared global memory. Each computer (or node) owns a processor and a local memory module. Inter-processor synchronization is achieved with a message-passing network.

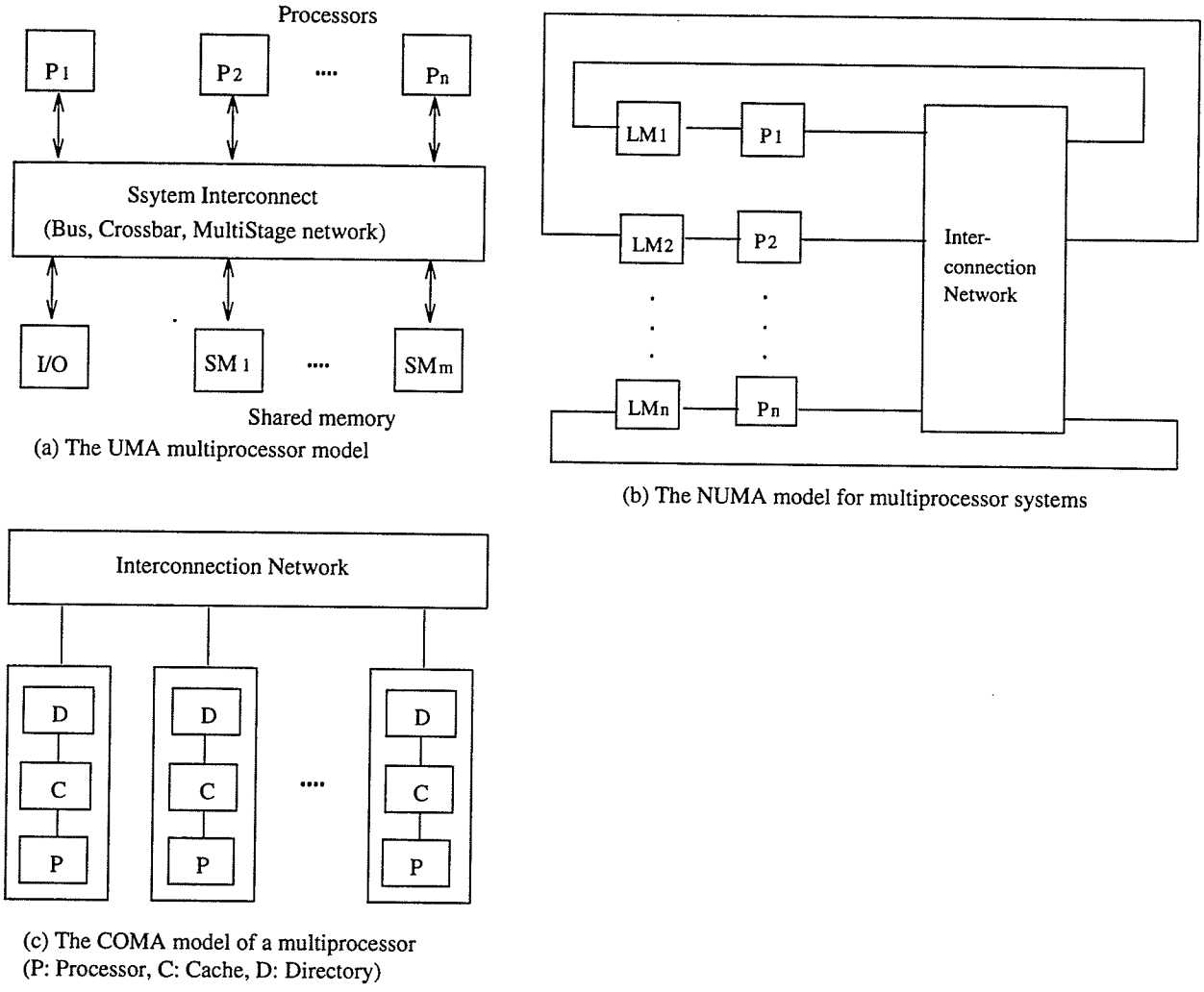


Figure 2.4: Shared-Memory Multiprocessors (quoted from [1], pg. 20-25)

Each node in a traditional message-passing network is connected to others through point-to-point static connection channels. Each node connects to a router uses the static connection network to send messages to other nodes possibly by cooperating with a sequence of routers and channels. Each local memory is accessed by its own processor and is not accessible by others. Therefore, these multicomputers are called *no-remote-memory-access* (NORMA) machines. However, some new generation parallel processors, such as SP2 [10], use a *multi-stage interconnection network* (MIN) instead of point to point static channels which provides a higher bisection bandwidth over the static channels.

Programming on a distributed-memory multicomputer is difficult because programmers must allow for distributed computation, data flow between nodes, and interprocessor communication. Building a sound parallel programming environment is critical if the difficulty of programming on multicomputer systems is to be feasible.

The fundamental shortcoming of shared-memory multiprocessors is they are not architecturally scalable. Distributed-memory multicomputers provide scalable architecture but offers a difficult programming model. The latter model offers a significant long term advantage but requires more research to make the computing environment acceptable for the program developers.

2.2 Parallel Programming

Code development on conventional uniprocessor computers requires programmers develop code for a sequential programming environment. The computation model

for such computers is based on SISD machine model. The instructions in a sequential program are executed one after another. Most existing languages, compilers, and operating systems are developed for uniprocessor based computers. When using a parallel computer, software developers want to exploit a parallel programming environment that detects parallelism in the program.

To provide a parallel programming environment, new language extensions or constructs must be developed that support parallelism. This first requires techniques to detect parallelism at various granularity levels, probably by using intelligent compilers. There are two popular parallel programming techniques:

- Using a conventional language, such as C or Fortran and a parallelizing compiler, a sequential program is translated into a parallel node program. The compiler must have the ability to detect and exploit parallelism in the sequential program and to distribute computation using the target machine resources (Figure 2.5(a)). This method has been applied to both shared-memory multiprocessors and distributed-memory multicomputers. The 'intelligence' of a parallelizing compiler is a decisive factor in the success of this approach. The major advantages are that: (1) no programmer retraining and (2) the existing conventional software does not need to be rewritten.

- Alternatively, programmers can be required to specify the parallelism in their programs (Figure 2.5(b)). The programmer needs to develop an explicit parallel program by using parallel dialects of C or Fortran . The compiler is less complicated than a compiler for the first method, because some of responsibility has been passed into the programmers. The compiler only needs to

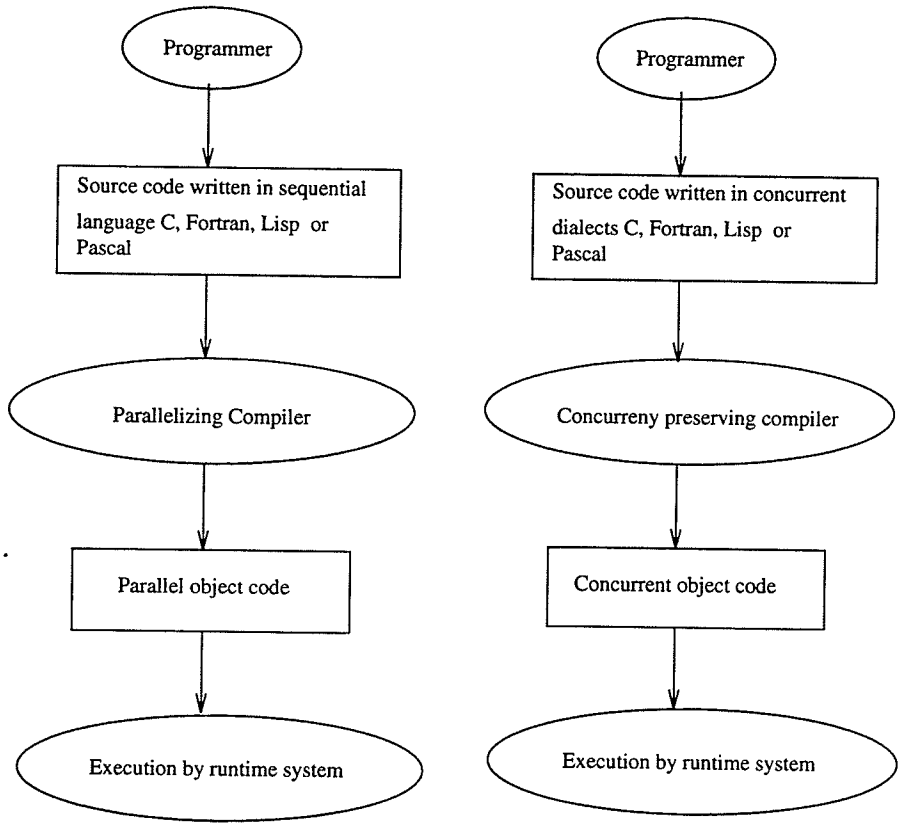


Figure 2.5: Two Approaches of Parallel Programming (quoted from [1], pg. 19)

preserve parallelism specified and generate parallel code.

Machine-Independent Parallel Programming

Despite all the advantages of parallel machines, programming on parallel machines is still widely believed to be very difficult. Programmers must learn the new parallel programming environment and to exploit the underlying machine architecture. Programs must be rewritten in a parallel programming language that reflects the underlying machine architectures. For example, a message-passing dialect for a MIMD distributed-memory machine, extended vector and array syntax for a SIMD machine, or an explicitly parallel dialect with synchronization for a MIMD shared-memory machine [13].

Most existing parallel programming models are machine-dependent so programmers need to deal with machine-specific issues such as improving data locality and the exploitation of specific memory hierarchies. Such parallel programs cannot be easily modified or ported to different machine architectures. It should be noted that programming MIMD distributed-memory machines is the most challenging environment because each processor has its local memory (address spaces) and inter-node communication is through calls to machine-specific communication libraries. Programmers must write their programs in a message-passing dialect and deal with address translation and processor synchronization using messages. The resulting programs are often nearly impossible to maintain.

One way to solve the parallel programming problem is by providing a machine-independent programming model. This model must be easy to program and be

portable. A good parallel language model should have the following attributes: (1) efficiency in its implementation, (2) portability across different architectures, (3) compatibility with existing conventional sequential languages, (4) expressiveness of parallelism, and (5) ease of programming.

2.3 Related Work

Before preceding to the thesis itself a brief review at some related work will help to set the framework. Although all issues may not be clear at this point the balance of this thesis will clarify the significance of this related work.

FORTRAN D can be viewed as a second-generation distributed-memory compiler. It incorporates and extends many compiler-time analysis and optimization techniques from many other research projects, which can be treated as the first-generation compilers. Some of the important ones are SUPERB, Kali, and CM FORTRAN [13].

SUPERB is a semi-automatic parallelization tool designed for MIMD distributed-memory machines [15, 13]. This tool provides user-specified BLOCK distributions and performs dependence analysis to guide program transformations and communication optimizations. SUPERB first uses guards and element-wise messages to generate an intermediate program using run-time resolution and then applies the transformations and optimizations to the intermediate code. It originates *overlaps* as a means to store nonlocal data accesses. SUPERB also provides interprocedural data-flow analysis of parameter passing. This tool does not support CYCLIC dis-

tribution, collective communications, dynamic data decomposition and storage of nonlocal values in temporary buffers.

Kali is the first compiler system which supports both regular and irregular distributions on MIMD distributed-memory machines[16]. It supports BLOCK,CYCLIC, and user-specified distributions. Kali requires the programmer explicitly to partition the loop iterations among processors by specifying an *on* clause for each parallel loop. Arguments to procedures are labeled with their expected incoming data distribution. It does not provide dependence analysis. Mandatory *on* clauses for parallel loops, collective communications, and dynamic decompositions are major differences between FORTRAN D and Kali.

CM FORTRAN is a version of FORTRAN 77 with vector constructs[17]. Programmers must explicitly specify data parallelism by using of vector operations or making array dimensions as parallel. CM FORTRAN allows users to define a data partition for each partition by using *interface blocks.* Array parameters are copied to buffers of the expected distribution at run time, eliminating the interprocedural analysis.

Compared with the first generation research projects, FORTRAN D has provided two main advantages: (1) the dependence analysis helps the compiler to exploit parallelism in the source code, without using any language extending constructs or explicitly specifying parallelism in the source code. The analysis also provides the information for compiler to apply more optimizations, (2) the FORTRAN compiler performs its analysis up front and use it to derive the SPMD code, decreasing the load of run-time system.

2.4 Summary

The existing parallel languages can be divided into two groups [1, 3]: (1) a real new parallel language and (2) a conventional programming language enhanced with new constructs. The advantage of a new language is providing explicit high-level constructs which present the computation models for parallel architectures, but new languages are often incompatible with existing languages. Most existing parallel languages belong to the second group [1].

In Chapter 3, the FORTRAN D compiler is introduced because it is a machine independent language. This language was developed at Rice U. and it adopted the second approach of parallel programming. It meets all the requirements of a good parallel programming language, described above and provides a suitable environment for demonstrating the new innovations proposed in this thesis.

Chapter 3

FORTRAN D

Currently several parallel architectures are used, each equipped with its own communication library. This introduces two difficulties: (1) programmers have to adapt to multiple programming environments and (2) it is difficult to port software across different platforms. One solution to this problem is a machine-independent programming model suitable for a various parallel architectures.

FORTRAN D is a version of FORTRAN enhanced with a rich set of *data decompositions* [13]. It provides a machine-independent programming model for distributed MIMD machines which supports new features that make it possible to write efficient machine-independent data-parallel programs using data decomposition specifications. FORTRAN D allows programmers to express parallelism but it is simple enough to let a compiler generate efficient SPMD programs for different parallel machine architectures. FORTRAN D also supports techniques such as automatic data decomposition and communication generation.

The balance of this chapter introduces and describes the FORTRAN D programming language. Since the focus of the discussion is on parallelism, a working knowledge of FORTRAN is assumed, so the chapter focuses on the extensions to support parallelism. Finally, section 3.2 makes few summary comments.

3.1 FORTRAN D Language

The data decomposition problem in a FORTRAN D program is solved using two levels of parallelism [8]. The first level is called *problem mapping* and decides how arrays should be aligned with respect to one another in the program. It is decided by the structure of the underlying computation, but is mostly independent of any machine architecture. Problem mapping is the basic requirement for reducing data movement during program execution. The second level is called *machine mapping.* It decides how arrays should be distributed onto the processors. When applying machine mapping, architectural characteristics such as the topology, communication mechanisms, size of the local memory on each processor, and number of processors in the system must be considered. Some program characteristics like the size of a distributed array and computation structure may also determine data distribution.

During loop execution in a parallel program, inter-processor communication primarily depends on mapping iterations and array elements to processors. Consider the program segment in Figure 3.1. Depending upon the different data distribution of the arrays and the scheduling of iterations on n processors it is possible to generate different communication overhead. For example, if A(i) and B(i+1) are mapped to the same processor and iteration i is executed on the same processor,

```
for i = 1, n
        A(i) = B(i+1)
end
```

Figure 3.1: Example of Problem Mapping

no communication is required to execute the iteration. However, if iteration i is executed on a different processor, then B(i+1) must be sent to the computing processor and the result must be stored back at the processor that owns A(i). Clearly, the first alternative is preferred. Now consider a case where A(i) and B(i+1) are on different processors. In this case, there are three different scenarios: (1) the i^{th} iteration is executed by the owner of A(i); (2) the i^{th} iteration is executed by the owner of B(i+1); (3) the i^{th} iteration is executed by a different processor. In first case, B(i+1) must be sent to the owner of A(i) to perform the computation. In the second case, the final result must be sent back to the owner of A(i). In the third case, B(i+1) must be sent to the computing processor and the final result must be sent back to the owner of A(i). From these examples it can be seen that most communication overhead is due to poor program mapping (i.e., the alignment of arrays does not match the computational requirements).

3.1.1 Problem Mapping

Problem mapping is very important in a FORTRAN D program because at the need to minimize the number of communication statements. Most parallel programs contain explicit or implicit communications caused by the memory hierarchy

of massively parallel processors. A "nonlocal" data access results in communication to a remote processor and it is a major factor in degrading the performance of a parallel system. It should be noted that it could be impossible to eliminate all communication so the goal is to minimize the quantity.

Problem Mapping Statements

Fortran D problem maps using *DECOMPOSITION* and *ALIGN* statements. A decomposition is simply an abstract problem or index domain and no storage is associated with it. The *DECOMPOSITION* statement declares a decomposition with its name, dimensionality, and size. Figure 3.2(a) illustrates two decompositions: D1 is an one-dimensional decomposition of size *N* and D2 is a two-dimensional decomposition of size *N*N*. The ALIGN statement is used to map arrays onto a decomposition. Arrays mapped to the same decomposition are aligned with each other. The alignment is specified by the placeholders in the subscript expression of the arrays and decomposition.

Each alignment inside a program can be classified as one of the following types: *exact match*, *intra-dimension alignment*, and *inter-dimension alignment*. Exact match is the simplest form of alignment (Figure 3.2(a)). It maps the array exactly onto the decomposition. In an exact match, no placeholder is required. Intra-dimension alignment specifies an offset and/or a stride for each subscript expression of the data decomposition. The programmer can assign an alignment and/or an alignment stride for any dimension of an array. An alignment stride is used as the coefficient of the placeholder in the subscript expressions of a decomposition. Figure 3.2(b) illustrates intra-dimension alignment with offsets, strides, or both for

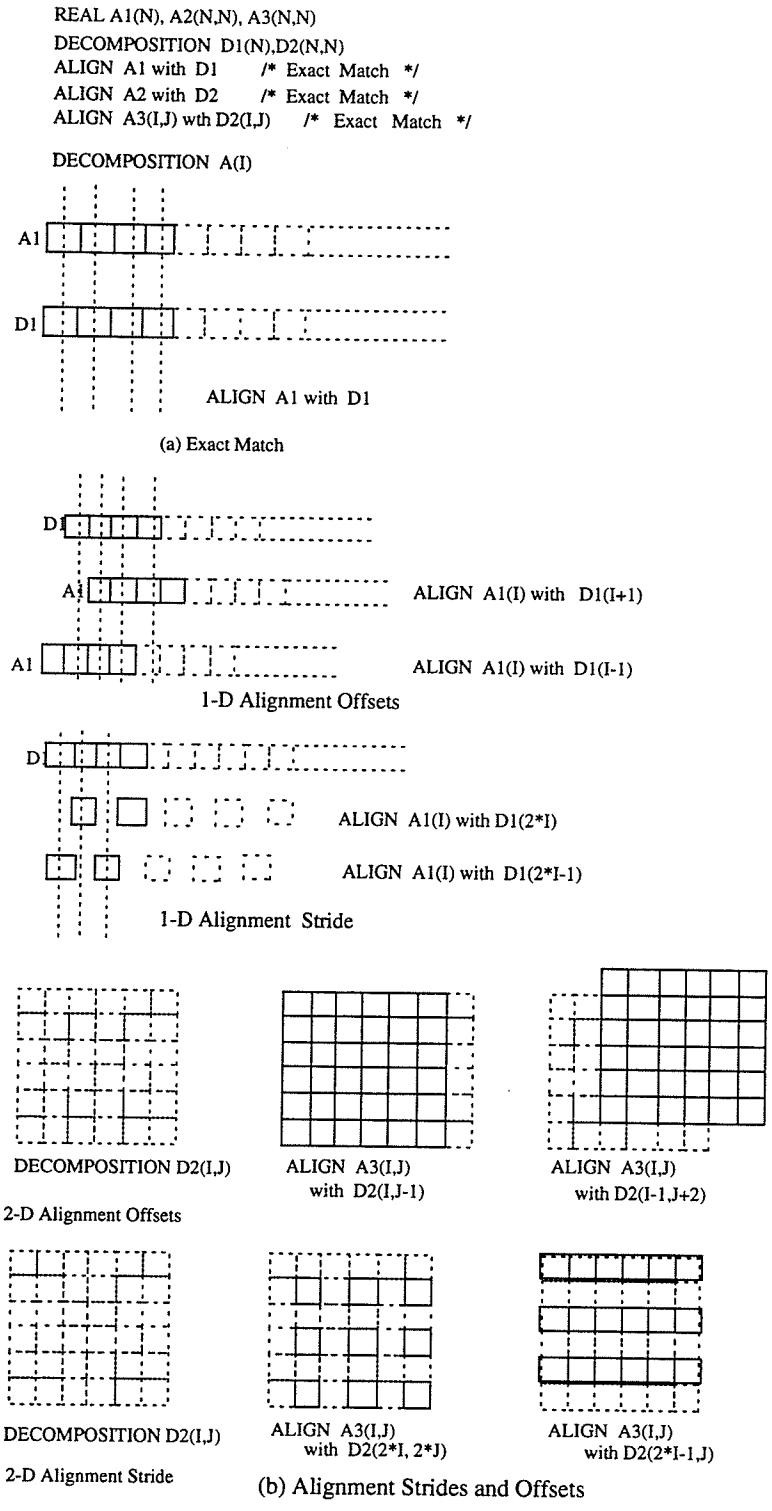


Figure 3.2: Example of Problem Mapping Statements

one-dimensional and two-dimensional arrays.

Inter-dimension alignment determines the data decomposition between dimensions. It includes *permutation*, *collapse*, *embedding*, or any combination of these three. *Permutation* permutes the dimensional alignment between arrays and decompositions. This can be done by exchanging the position of placeholders in the decomposition, such as "ALIGN A3(I,J) with D2(J,I)". *Collapse* omits the assignments of certain dimensions of the array while mapping the array onto its decomposition. All the data elements in the omitted dimensions are collapsed and assigned to the same location in the decomposition. *Array embedding* is the inverse of array collapsing. It maps arrays with fewer dimensions onto a higher dimensional decomposition. Each unmapped dimension of the decomposition is given an explicit value to denote the position of the mapped dimension(s). Any combination of intra-dimensional and inter-dimensional alignments is allowed when applying array-to-decomposition mapping. Figure 3.3 illustrates array permutation, array collapse, and array embedding.

FORTRAN D also provides some useful options for the ALIGN statement: *overflow*, *range*, and *replication*. The *overflow* option specifies the actions to be taken when an array-to-decomposition overflow occurs. If it happens, the user can select a method from three default actions: *ERROR*, *TRUNC*, and *WRAP*. These options are illustrated in Figure 3.4. FORTRAN D permits mapping at part of an array onto its decomposition using the *range* option of the *ALIGN* statement. The *range* of a dimension is specified by the from:to syntax. The symbol "*" is used to denote the entire array dimension should be mapped. The final option is *replication* which replicates distributed variables into a decomposition. The range option can be used

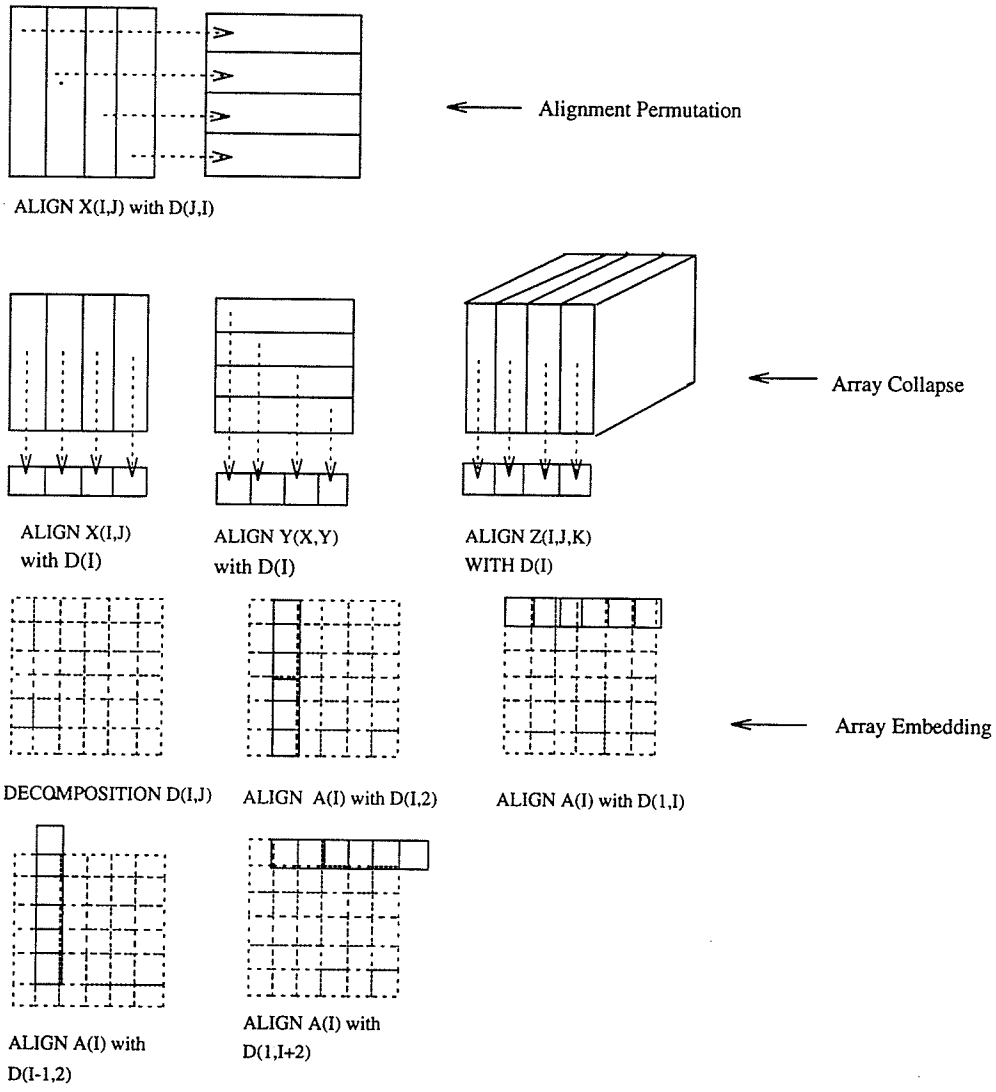


Figure 3.3: Example of Array Permutation, Collapse, and Embedding

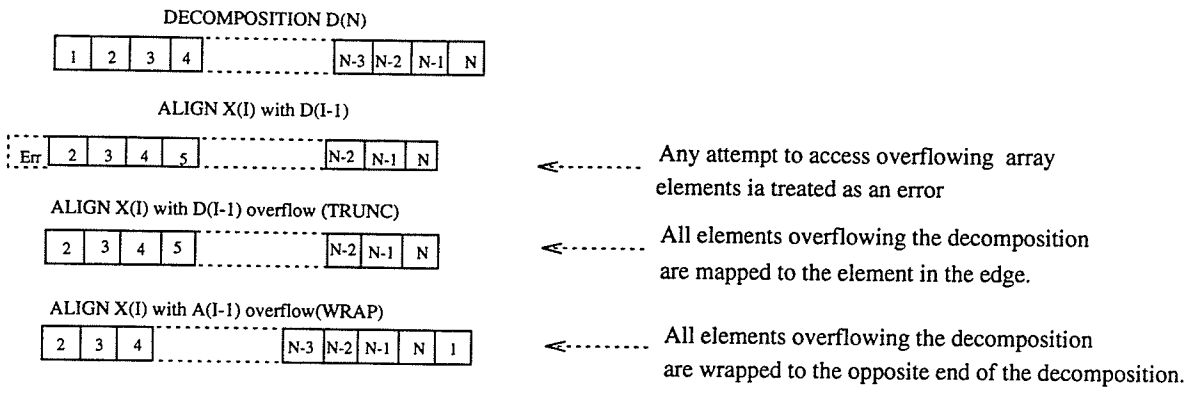


Figure 3.4: Example of Alignment Options

with the replication option.

3.1.2 Machine Mapping

ALIGN and DECOMPOSITION statements together specify how different data structures should be aligned. These two constructs provide a mechanism to reduce interprocessor communication due to unaligned data structures. However, interprocessor communication may still be needed if poor iteration mapping occurs. FORTRAN D's DISTRIBUTE statement specifies data distribution onto the processors. This information is used to schedule iterations so interprocessor communication is minimized.

Each FORTRAN D's decomposition variable can only be specified with one distribution. The compiler will apply the distribution to arrays aligned with this decomposition and defines data distribution for each dimension in the decomposition. The format of the DISTRIBUTE statement is:

DISTRIBUTE D1(*attribute*)
DISTRIBUTE D2(*attribute,attribute*)

There are three types of attributes in FORTRAN D: *BLOCK*, *CYCLIC*, and *BLOCK_CYCLIC*. BLOCK distribution divides the size of decomposition into contiguous blocks of size (N/P), where N is the size of the decomposition and P is the number of processors. The compiler will then assign one block to each processor. CYCLIC adopts a round-robin distribution for the decomposition. It assigns every P^{th} element to the same processor which is good for load-balancing. BLOCK_CYCLIC is a combination of the above two distributions which takes a parameter X representing the block size (M). After dividing the decompositions into contiguous chunks of size M, the compiler applies the CYCLIC distribution to distribute these blocks. Figure 3.5 has the examples of these three types of distribution.

FORTRAN D also allows the user to define the number of processors for each dimension of a decomposition. Figure 3.6 illustrates this data distribution where each distributed dimension is given a number of processors on that dimension.

To support programs with irregular data parallelism, FORTRAN D provides irregular distributions. The user can assign the distribution through a mapping array which itself will be distributed. Figure 3.7's program segment illustrates irregular distribution where the distribution of each element in decomposition IRREG is determined by the value of its corresponding array element in MAP. The mapping result of IRREG on each processor is shown in Figure 3.8.

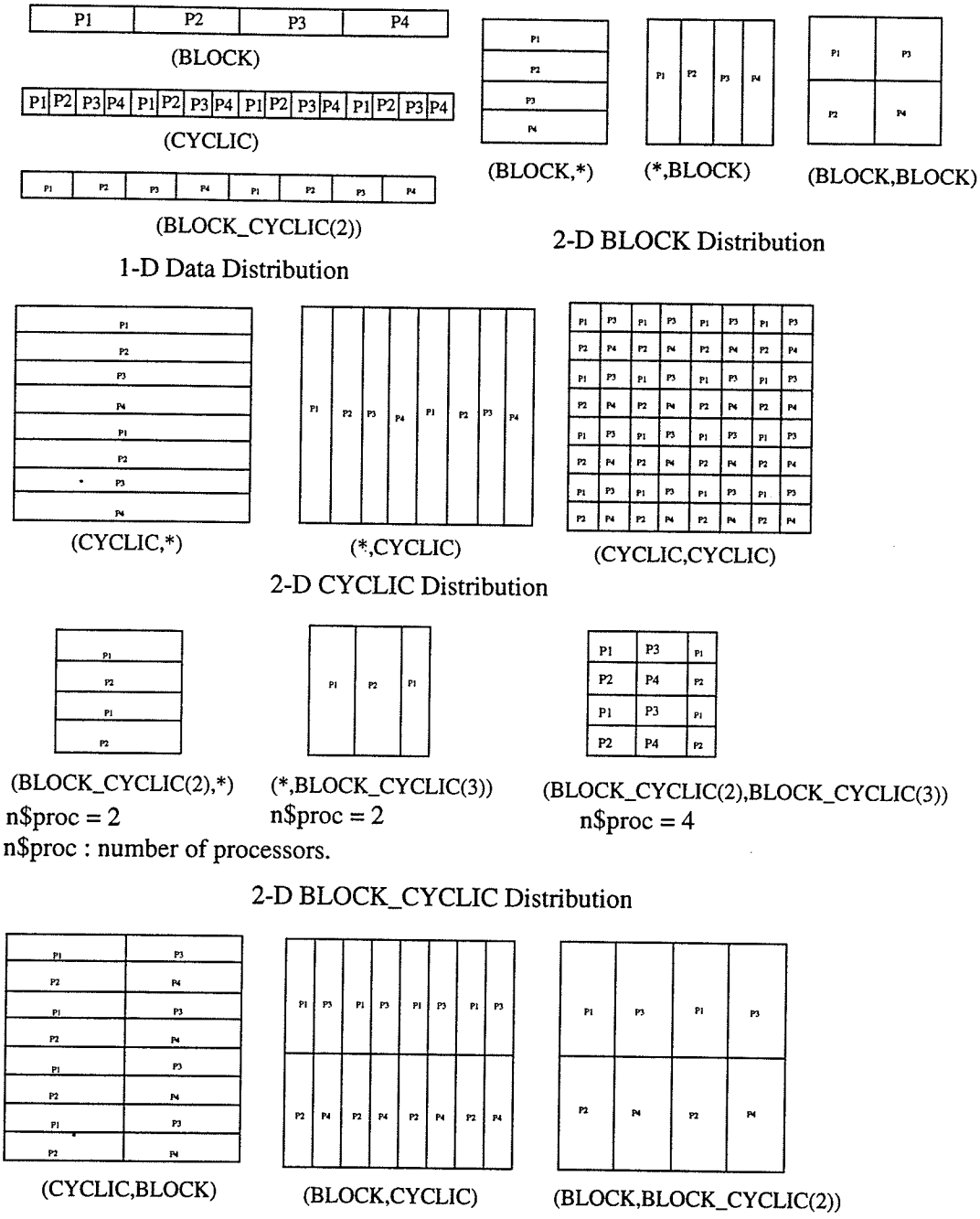


Figure 3.5: Example of Data Distribution

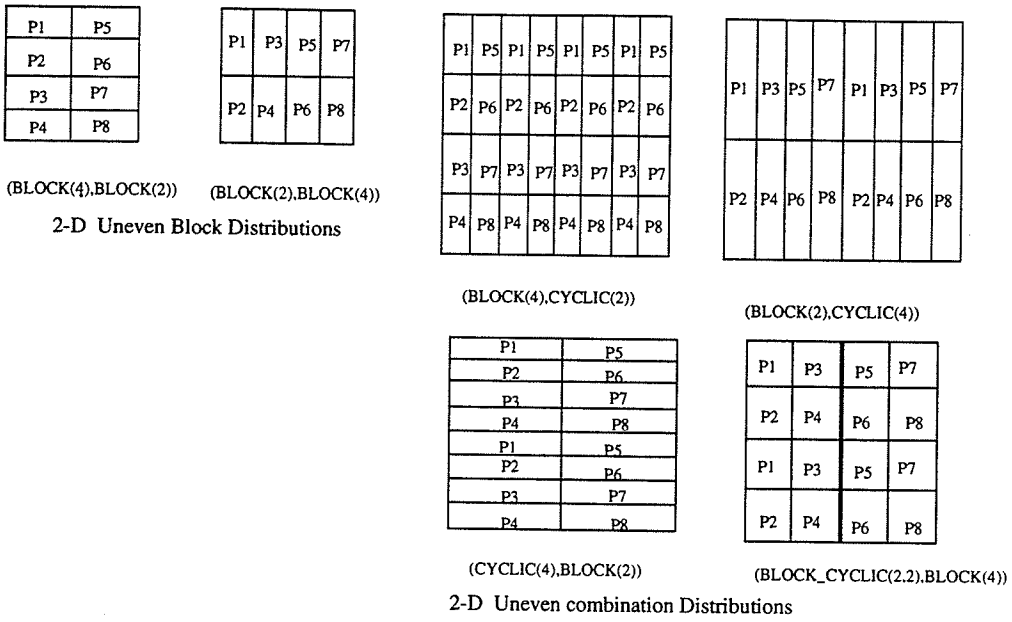


Figure 3.6: Example of Processor Distribution

3.1.3 . Additional Features of FORTRAN D

The following additional features are available in FORTRAN D [8] too:

Dynamic Data Decomposition: The computation pattern may change between different phases of the program. One way to reduce data movement is to change data alignment between the different phases. FORTRAN D applies dynamic data decomposition to provide this ability. Each ALIGN and DISTRIBUTE statements inside the program may be interpreted as executable statements, rather than declarations, so the data distribution can be changed dynamically during program execution.

```
n$proc = 4
REAL X(16)
INTEGER MAP(16)
DECOMPOSITION EG(16), IRREG(16)
ALIGN MAP with REG
ALIGN X with IRREG
DISTRIBUTE REG(BLOCK)
.....set values of MAP array
DISTRIBUTE IRREG(MAP)
```

Figure 3.7: Example of Program Segment with Irregular Distribution

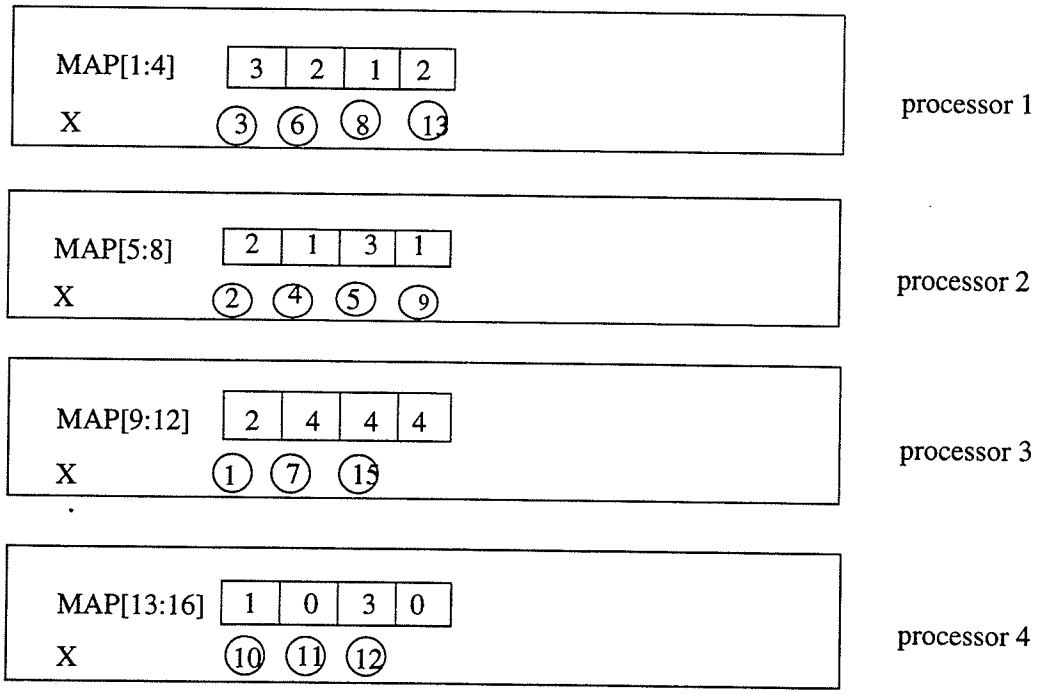


Figure 3.8: Example of Irregular Distribution

```
FORALL I = 1, N
      X(INDEX(I)) = ...
      ... = X(INDEX(I+1))
ENDDO
```

Figure 3.9: Example of FORALL Loop

Procedures: The scoping rule for data decompositions in FORTRAN D procedures is defined as: (1) procedures inherit data decompositions from their callers, and (2) the effects of all *DECOMPOSITION*, *ALIGN* and *DISTRIBUTE* defined inside a procedure are limited to itself and procedures called by it. A distributed array can also be passed as a parameter.

FORALL loops: FORALL loops prevent synchronization in a loop whose dependences can not be detected at compile-time (eg. using index arrays). FORALL loops can only use values defined before entering the loop or within the current iteration, so each iteration has its own copy of the entire data space before entering the loop. Therefore, all the iterations in the loop can be executed in parallel without communication. At the end of a FORALL loop, the compiler performs another important operation - *merge*, which is used to merge the variables that are assigned new values by different iterations during the loop. The merge operation is done in a deterministic way using values assigned from the latest sequential iteration. Figure 3.9 is a example for FORALL loop where each iteration of the FORALL loop uses the values of array elements, INDEX(I) and INDEX(I+1) defined before the loop.

Reduction: A *reduction* can be performed on a group of data. The result for the operation will have lesser dimensionality or just a single scalar value. The *REDUCE* statement is used to specify the *reduction* operation in a FORTRAN D program. The following reduction functions are provided by FORTRAN D:

SUM	sum of a list of numbers
PROD	product of a list of numbers
MIN	minimum of a list of numbers
MAX	maximum of a list of numbers
AND	logical AND of a list of booleans
OR	logical OR of a list of booleans

The programmers are also allowed to define their own reduction functions if they are associative and commutative.

ON clause: FORTRAN D allows user to specify the processor for each iteration of the loop. This feature permits the users to control load-balancing and reduce communications.

3.2 Summary

The principal goal of Fortran D is to provide support for exploiting fine-grain parallelism on distributed-memory machines. It provides a simple programming model for the programmers through the support of a sophisticated compiler. FORTRAN D

provides high compatibility with FORTRAN. After removing the data decomposition statements in a FORTRAN D program, it becomes a FORTRAN program. To develop a sound parallel programming environment for distributed-memory MIMD machines, several open questions must be addressed: automatic data decomposition, static performance estimation, run-time preprocessing using the PARTI communication library, additional high-level language constructs to support parallel operations, etc [13]. This thesis now turns its attention to the FORTRAN D compiler.

Chapter 4

FORTRAN D Compiler

Two steps are required to write a data-parallel program in FORTRAN D: (1) selecting a suitable data decomposition and (2) using it to derive the *single program over multiple data* (SPMD) program with explicit communications to access non-local data. The first part is the duty of the programmer and the second is the FORTRAN D compiler.

The FORTRAN D compiler translates a program into a SPMD node program with explicit message passing for distributed-memory machines. The compiler detects and exploits parallelism in a FORTRAN D program and produces a node program with minimal communication overhead executable on the nodes of a MIMD machine. The FORTRAN D compiler uses the principle that the *owner* computes to generate the node program of the source program. Therefore, each processor only performs the computation for its own local data set so means each processor can only compute the values of data set assigned to it by data distribution. The data decomposition specifications in the source program are changed into mathematical

distribution functions by the compiler that are used to determine the ownership of local data. FORTRAN D compilation can be roughly divided into three major phases: *program analysis* (section 4.1), *program optimization* (section 4.2), and *code generation* (section 4.3).

4.1 Program Analysis

There are four parts to program analysis: *dependence analysis*, *data decomposition analysis*, *partition analysis*, and *communication analysis* [13].

4.1.1 Dependence Analysis

This phase analyzes the control flow and memory accesses in the source program to determine a statement execution order that produces the same result as the original program. If there is a *data dependence* between two references *Rl* and *R2* in a source program, then both reference the same memory location. To preserve the semantics of the original program, their execution order must be consistent with the sequential execution order. If there is a dependence between *R1* and *R2* and *R1* must be executed before *R2*, then *Rl* is called the *source* of this dependence and *R2* is its *sink*. Data dependences define these ordering relationships in a source program. Four kinds of data dependences exist:

Flow (True) dependence: If the source *R1* is a *write* operation and *R2* is a *read.*

Antidependence: If the source *R1* is a *read* operation and *R2* is a *write.*

Output dependence: If both source *R1* and sink *R2* are *write* operations.

Input dependence: If both source *R1* and sink *R2* are *read* operations. Input dependences do not restrict statement order.

Each data dependence in a loop identifies two types of dependence relations. Consider the sequential loop nest L in Figure 4.1, where I_i, p_i, and q_i, respectively, are the index variable, lower bound, and upper bound of L_i for $1 \leq i \leq m$, and $H(I_1,I_2,...,I_m)$ is the loop body. This loop nest L is denoted as $L = (L_1,L_2,...,L_m)$. When m is the arity of the loop, so if m = 3 it is a triple-nested loop. The iterations of L are represented by the index vector of the loop, denoted by $(I_1,I_2,...,I_m)$. The index values of L are the values of the integer vector $(i_1,i_2,...,i_m)$, such that $(p_1,p_2,...,p_m) \leq (i_1,i_2,...,i_m) \leq (q_1,q_2,...,q_m)$, respectively. Each index value $(i_1,i_2,...,i_m)$ generates an instance of the loop body, denoted by $H(i_1,i_2,...,i_m)$, which is an iteration of L. If there is a data dependence from an instance *S(i)* of S to an instance *T(j)* of T where S and T are two statements of the loop body, we can say "T depends on S" based on the dependence relation between statements. Similarly, "*T(j)* depends on *S(i)*" based on the relation between the iterations of the loop. A data dependence in L can be characterized by its *dependence distance* and *level.* The dependence distance of a dependence from *S(i)* to *T(j)* is defined as j - i. Its dependence level has $m+1$ possible values: *1,2,..,m+1* which is decided by the first non-zero entry in the distance vector. If T depends on S at a level l, $1 \leq l \leq m$, then the dependence of T on S is carried on loop L_l. The dependence of T on S is loop-independent if it is not carried by any loop (i.e, the distance is 0 and the level is $m+1$).

```
L1: do I1 = p1, q1
L2:   do I2 = p2, q2
        ...
Lm:       do Im = pm, qm
                H(I1,I2,...,Im)
            enddo
        enddo
    enddo
```

Figure 4.1: Sequential Loop Nest

To provide a correct and efficient dependence test is important for a parallelizing compiler. In the dependence analysis phase, the compiler must compute all the data dependences in a program. Dependence testing is a method used to decide whether dependences exist between two subscripted references to the same array in a loop nest. To calculate the data dependences due to array references in a loop nest, the compiler must solve a set of linear equations in the integer space [9]. Each dependence found is represented by its dependence distance and level. This information is crucial in guiding a FORTRAN D compiler during optimization . The FORTRAN D compiler is developed in the context of the ParaScope programming environment which incorporates this analysis [13, 14].

4.1.2 Data Decomposition Analysis

In this phase, the compiler determines the data decomposition at each reference of a distributed array. A particularly important feature of FORTRAN D is dynamic

data decomposition. Although this feature provides more flexibility, it makes the job of the FORTRAN D compiler more difficult. To generate the correct guards (conditional statements) and communication in the node program, the compiler must know the correct decomposition for each array reference at any point of the program. *Reaching decompositions* are the possible set of data decomposition specifications that may reach an array reference. The compiler applies both intra- and interprocedural analysis to decide the data decompositions for each reference to a distributed array. If multiple decompositions reach a procedure, node splitting or run-time techniques are applied to generate the correct code for the program [13].

The scope rules for dynamic data decompositions in a FORTRAN D procedure are:

1. a procedure inherits data decompositions from the callers, and
2. the effects of dynamic data decompositions defined inside a procedure are limited to itself and procedures called by it.

Since the dynamic data decompositions defined inside the procedure are not valid after the procedure returns, the compiler must insert appropriate (and potentially expensive) calls to run-time data distribution routines to restore the original data decomposition.

4.1.3 Partitioning Analysis

Program partition analysis uses *data decomposition specifications* and *the owner computes* rule to partition the data and computations of the original program among

processors. The compiler first uses data decomposition specifications to partition the arrays to processors and then uses the *owner computes* rule to distribute the computation onto the processors. *Iteration sets*, *index sets*, and *regular section descriptors* (RSDs) are used during this phase. An iteration set is composed of loop iterations that represents a section of the work space. An index set is a set of array indices that represents a section of the data space. The FORTRAN D compiler uses regular section descriptors as internal representation for iteration and index sets. The representation of an RSD is $[l_i{:}u_i{:}s_i,...]$ where l_i, u_i, and s_i respectively represent the *lower bound*, *upper bound*, and *step* of the i^{th} dimension of the RSD. If a step of a RSD is not explicitly specified, then it is assigned the default value of one. This representation provides an easy way for the compiler to describe rectangular or right-triangular array sections. Operations, such as intersection and union, can be easily performed on RSDs. In an RSD representation of a nested loop or a multidimensional array, the leftmost dimension of the RSD maps to the outermost loop of the loop nest or the leftmost array dimension of the array, respectively. In most cases, RSDs are used in array section analysis to record the array sections defined by an assignment statement at the loop level. The compiler can use the RSDs computed for communication optimizations. Consider the example in Figure 4.2. The compiler computes a RSD for the reference A(i,j) at each loop level and then tags it on that loop level for later use. In this example, the RSDs for each loop level starting from the innermost level are A[i,j], A[i,1:100], and A[1:100,1:100].

The FORTRAN D compiler produces a SPMD program that can be executed directly on each node of a MIMD machine. It means that all the processors must execute the same program and have the same array declarations in the node program. Since all the processors are using the same array declarations, they must have

```
do i = 1, 100                          /* RSD = A[1:100,1:100] */
      do j = 1, 100                    /* RSD = A[i, 1:100] */
            A(i,j) = B(i,j).....       /* RSD = A[i, j] */
      enddo
enddo
```

Figure 4.2: Example of RSD Representation

```
{* Original program *}                {* SPMD node program *}
REAL A(40)                            REAL A(10)
do i = 1, 40                          do i = 1, 10
      A(i) = 5.0                            A(i) = 5.0
enddo                                 enddo
```

Figure 4.3: Example of Global and Local Indices

a *local-index* set which is different from the *global indices* in the original program. The example in Figure 4.3 illustrates one-dimensional array (A) of the original program that is block-distributed among four processors. The SPMD node program has different array and loop indices from the original program. In the node program, the local indices for A are [1:10] but the global indices for array A after the data partition are [1:10], [11:20], [21:30], and [31:40]. Therefore, the loop indices in the node program are [1:10] but the global loop indices in the FORTRAN D program are [1:40].

The way that FORTRAN D compiler computes the local index set is almost the same. The compiler first decides the reaching decomposition set for each reference

```
REAL A(100,100), B(100,100)
DECOMPOSITION D(100,100) {* The number of processors on the machine is four. *}
DISTRIBUTE D(:,BLOCK)
do k = 1, time
    do j = 2, 99
        do i = 2, 99
S1:         A(i,j) = (B(i,j-1) + B(i-1,j) + B(i+1,j) + B(i,j+1))/4
        enddo
    enddo
    do j = 2, 99
        do i = 2, 99
S2:         B(i,j) = A(i,j)
        enddo
    enddo
enddo
```

Figure 4.4: Example of a Fortran D Program(Jacobi)

of a distributed array, then according to the reaching decomposition it computes the local index set of each array which is the local array section owned by each processor. Consider the Jacobi program shown in Figure 4.4 [13] where both arrays A and B have the same data distribution as the data decomposition variable D. The first dimension of D is undistributed and the second dimension is block-distributed. The reaching decomposition set for both arrays has only element D and the local index set on each processor is [1:100,1:25].

The computation of the local index set decides the data partition for each array. By using the result of data partition phase the compiler partitions the computation

among the processors using the *owner computes* rule. The *local iteration set* of a reference R on a processor t_p is defined to be the set of loop iterations that cause R to access data owned by t_p. This iteration set can be calculated in two steps [13]. First, apply the inverse of the array subscript function of R to the local index set of t_p, and second intersect the result of step one with the iteration set of the enclosing loops. The FORTRAN D compiler only computes the local iteration set of *left-hand side(lhs)* for each statement, because of the *owner computes* rule. Therefore, the set of iterations to be executed on a processor is given by the union of local iteration sets of the *lhs*s of each assignment statement.

The Jacobi program in Figure 4.4 helps explain how to compute the *local iteration set.* The compiler applies the inverse of the subscript function of the *lhs* to the local index set of A (i.e, [1:100,1:25]) and generates the unbounded local iteration set [:,1:25,1:100]. The first entry of this set, ":", indicates that every iteration of the k loop accesses local elements of A. The inverse subscript function maps j and i loops to [1:25] and [1:100], respectively. In the second phase, the compiler determines the intersection of the unbounded iteration set and the actual bounds of the enclosing loops, since these are the only iterations that actually exist. In Figure 4.4, the global iteration set of the loop is [1:*time*,2:99,2:99]. The local iteration sets for statement S1 in Figure 4.4 which are computed by two steps: (1) converting the global iteration set into the local iteration sets for each processor and (2) performing the intersection with the unbounded local iteration sets for each processor (in local indices) are:

The local iteration set for $Proc(1)$ = [1:*time*,2:25,2:99]
The local iteration set for $Proc(2{:}3)$ = [1:*time*,1:25,2:99]

The local iteration set for $Proc(4)$ = [1:*time*,1:24,2:99]

The same procedure is applied to compute the *local iteration sets* for statement S2. This algorithm uses the local indices - calculated for the local index set of each array - to derive the local indices for the local iteration set. To generate a SPMD node program which can be executed on all the processors, the FORTRAN D compiler must ensure that the processors on the boundary are different from the remaining (inner) processors because they may have different local index and iteration sets.

The third aspect to be considered at this stage is how to handling *boundary conditions*. The compiler handles boundary conditions differently when generating local index and iteration sets. In the program in Figure 4.5, the compiler will store different boundary conditions for processors one and four. This eliminates the need to calculate and store the condition for each processor. For each loop or array dimension, the boundary conditions for iteration or index sets are recorded in *pre*, *mid*, and *post* sets by the compiler [13]. The *pre* and *post* sets are for boundary conditions. These sets are represented as augmented iteration sets in the FORTRAN D compiler. Each dimension in an augmented set contains these three sets and their associated processors. As shown in Figure 4.5, the associated processors in the augmented sets are indicated by the '@' sign. The iteration set of a processor is the cartesian product of the corresponding iteration sets of each dimension. However, not all boundary conditions can be successfully represented by the augmented iteration sets. In a worst case, the compiler has to compute and store the index and iteration sets for each processor.

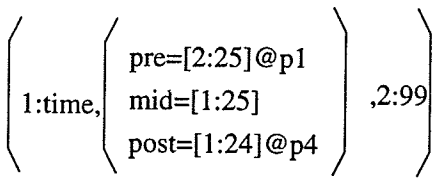


Figure 4.5: The Augmented Iteration Sets of Jacobi Program

```
do j = 2, 100
      do i = 2, 100
S1:       X = A(i,j+1) * B(i,j)....
S2:       A(i,j) = A(i,j) + 0.75 * (X-A(i,j))
      enddo
enddo
```

Figure 4.6: Example of Scalar Variables

The compiler replicates scalar variables on all processors. If the partitioning of a loop that assigns a value to a scalar variable is done using the *owner computes* rule, every processor has to execute all the loop's iterations. However, if the scalar variable is 'private', (used) within a loop iteration, each processor need not compute all the iterations. The compiler uses this fact and identifies the local iteration set for such an assignment as the union of the local iteration sets of the statements that use the scalar variable. For example, in the code segment of Figure 4.6, X is a replicated scalar that is identified as a *private* variable. The FORTRAN D compiler computes the local iteration set of S2 and assigns it to the local iteration set of S1.

4.1.4 Communication Analysis

In this phase the compiler uses local index and local iteration sets to compute nonlocal data accesses for each *rhs* reference. To compute the nonlocal index sets, all *rhs* references to the distributed arrays are examined. For each *rhs* reference, the compiler computes the RSD of index set to be accessed by each processor by applying the subscript functions of the *rhs* to the local iteration set assigned to the statement. Nonlocal data accesses are obtained by subtracting the RSD of local index set from the resulting RSD. The RSD corresponding to the nonlocal accesses will be retained only if it is not empty. For the boundary processors, the compiler must compute the nonlocal index set for each group of processors. For the Jacobi program (Figure 4.4), the compiler computes the *nonlocal index sets* for the three different processor groups. The local iteration set of the group of interior processors, Proc(2:3), is [l:time,1:25,2:99]. The local index set for B on processors Proc(2:3) is [1:100,1:25]. The nonlocal index set for each reference is shown in Figure 4.7. The *rhs* references B(i,j-1) and B(i,j+1) access nonlocal locations [2:99,0] and [2:99,26], respectively. Both references are marked and their nonlocal index sets are stored. Similar analysis produces the nonlocal index sets for the other two groups, Proc(l) and Proc(4). The nonlocal index sets for both processors are [2:99,26] and [2:99,0], respectively. For the rhs reference to A (i,j) in statement S2, the nonlocal index set is empty because only local accesses occur.

4.2 Program Optimization

The results of program analysis are used to improve parallelism and reduce communication overhead by detecting the opportunities for parallelism. Several advanced

```
The index set accessed by B(i,j-1) is [2:99,0:24]
The index set accessed by B(i-1,j) is [1:98,1:25]
The index set accessed by B(i+1,j) is [3:100,1:25]
The index set accessed by B(i,j+1) is [2:99,2:26]
The nonlocal index set for B(i,j-1) is [2:99,0]
The nonlocal index set for B(i-1,j) is []
The nonlocal index set for B(i+1,j) is []
The nonlocal index set for B(i,j+1) is [2:99,26]
```

Figure 4.7: Example of Index Sets and Nonlocal Index Sets for S1 in Jacobi Program

optimizations have been developed for the FORTRAN D compiler to exploit parallelism in a pipeline computation. The next chapter details optimizations for pipeline computations and communication , so only the fundamental concepts are introduced in this section.

Message Vectorization can reduce the communication overhead due to inefficient messaging. Consider the *run-time resolution* in Figure 4.8(b) for the program in Figure 4.8(a). The *send* and *recv* operations directly precede each reference causing a nonlocal data access. This simple approach is inefficient because it generates many small messages resulting in high communication overhead. The FORTRAN D compiler applies message vectorization to combine several small messages into a large message (see Figure 4.8(c)). The compiler uses data dependence information to combine small messages from inner loops into one large message that may be sent from some outer loop.

The compiler determines the appropriate loop level (called *commlevel*) to insert

```
{* Original program *}
REAL A(100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN X with D
do i = 1, 95
      A(i) = A(i+2)
enddo
```

(a) FORTRAN D Program

```
{* Run-time Resolution *}
REAL A(100)
my$p = myproc() {* 0...3 *}
do i = 1, 95
      if (my$p .eq. owner(A(i+2))) then
            send(A(i+2), owner(A(i)))
      endif
      if (my$p .eq. owner(A(i))) then
            recv(A(i+2), owner(A(i+2)))
            A(i) = A(i+2)
      endif
enddo
```

(b) Run-Time Resolution

```
{* Message Vectorization *}
REAL A(27)
my$p = myproc() {* 0...3 *}
if (my$p .gt. 0) send(A(1:2), my$proc-1)
if (my$p .lt. 3) recv(A(26:27), my$proc+1)
ub$1 = min((my$p+1)*25,95) - (my$p*25)
do i = 1, ub$1
      A(i) = A(i+2)
enddo
```

(c) SPMD Node Program

Figure 4.8: Example of Run-time Resolution and Message Vectorization

messages for nonlocal references. A loop-carried dependence's *commlevel* is defined as the level of the dependence. A loop-independent dependence's *commlevel* is defined as the level of the deepest loop common to both the source and sink of the dependence. Each *rhs* reference R with a nonempty nonlocal index set requires that the cross-processor flow dependences with R as the sink be identified as a loop-carried or loop-independent dependence. The deepest *commlevel*, L, of all such dependences is selected. If the dependence is loop-carried, the compiler tags R as loop-carried at the header of the loop corresponding to the deepest *commlevel*. If the dependence is loop independent and there exists a loop at level $L+1$ enclosing R, where L is the *commlevel* for the dependence, then R is tagged at the header of the loop at level $L+1$ as independent. R itself is tagged as independent at the statement containing it if none of the loops at level $L+1$ enclose it. A *carried* tag at L implies that nonlocal data accessed by R must be communicated between iterations of the loop L. An independent tag indicates that nonlocal data accessed by R must be communicated at this point on each iteration of loop L. The compiler may also move the *independent* tag to any statement in loop L between the source and sink of the dependence to combine messages from different references.

In the program of Figure 4.8(a), the *rhs* reference A(i+2) has nonlocal data accesses. The message for this reference is of type loop-independent and its *commlevel* is 2, so the compiler inserts a *independent* tag at the i loop because it is the next deeper loop enclosing A(i+2). Recall the Jacobi program of Figure 4.4, the *rhs* references B(i,j-1) and B(i,j+1) have nonempty nonlocal index sets. These references participate in the cross-processor true dependences. The source of this dependence is statement S2 (Figure 4.4). These dependences are *carried* at loop k. The compiler inserts a carried tag at the header of the k loop to signify the associated loop-carried

dependences. In the code generation phase, the messages for the references will be generated immediately following the k loop.

4.3 Code Generation

Code generation uses information collected during program analysis and optimization to generate the SPMD node program. Two phases are required: *program partitioning* and *message generation.* The steps in the program partitioning phase can be further divided into: *data partitioning*, *loop bound reduction*, and *guard introduction.* Figure 4.9 lists the SPMD code generated for the program in Figure 4.4.

The FORTRAN D compiler *partitions data* by reducing the array bounds of A and B so that each processor allocates storage only for local data. Then the compiler *reduces loop bounds* so that each processor only executes iterations in the union of local iteration sets of the *lhs* for each statement in the loop. In the final phase of code generation, the compiler *introduces guards* (inserts "if" statements) to: (1) handle the boundary conditions and (2) test the membership of a statement in the local iteration set.

Message Generation is the second step of code generation. The compiler uses information gathered during the analysis and optimization phases to generate messages. Figure 4.10 illustrates how the compiler inserts nonblocking *sends* and blocking *recvs* to handle messages. Arrays A and B are assumed to be block-distributed. Messages can be categorized as *loop-independent* and *loop-carried.*

```
REAL A(100,25), B(100,0:26)
if (P_local = 1)     lb1 = 2 else lb1 = 1
if (P_local = 4)     ub1 = 24 else ub1 = 25
do k = 1, time
if (P_local > 1)     send(B(2:99,1),P_left)
if (P_local < 4)     send(B(2:99,25),P_right)
if (P_local < 4)     recv(B(2:99,26),P_right)
if (P_local > 1)     recv(B(2:99,0),P_left)
      do j = lb1, ub1
         do i = 2, 99
               A(i,j) = (B(i,j-1) + B(i-1,j) + B(i+1,j) + B(i,j+1))/4
         enddo
      enddo
      do j = lb1, ub1
         do i = 2, 99
               B(i,j) = A(i,j)
         enddo
      enddo
enddo
```

Figure 4.9: The SPMD Code of Jacobi Program

Loop-Independent Messages: The messages for loop-independent references are tagged as: (1) *loop headers* where the compiler inserts code to *send* and *recv* primitives preceding the loop headers or (2) *individual references* where the compiler inserts *send* and *recv* primitives in the body of the loop preceding the reference. In Figure 4.10, the message for reference B(i+1) is a loop-independent message. The compiler tags the message for this reference at the k level so the communication primitives for this loop-independent message is inserted preceding the loop header. The compiler also introduces the guards for all the messages to ensure that the owner and the recipients execute *send* and *recv*, respectively. Each message has an associated RSD that represents the data sent on each iteration.

Loop-Carried Messages: The FORTRAN D compiler constructs an RSD for each *rhs* reference at the level of the loop carrying the dependence. The messages for *loop-carried* dependences can be classified as: *carried-all* and *carried-part.* For carried-all messages, the iterations of L are executed by all processors. The compiler inserts calls to *send* and *recv* primitives inside the loop header for L at the beginning of the loop body. In Figure 4.10, the messages for A(i+1) is of type carried-all at the k loop, so communication is inserted at the head of the loop body. For carried-part messages, the iterations of L are partitioned across processors. In this case, loop-carried messages represent data synchronization. The compiler inserts calls to *recv* preceding loop L, since they are executed before the local iterations of L. In Figure 4.10, the message for A(i-1) is of type carried-part at the i loop, so communication is inserted before and after the i loop header.

When the generated code contains both independent and carried-part messages at the loop header, the compiler orders the insertion of *send* and *recv* primitives rel-

```
{* Example Computation *}
do k = 1,M
      do i = 1,N
            A(i) = B(i+1)          {*commlevel is 3 *}
                 + A(i+1)          {*commlevel is 1 *}
                 + A(i-1)          {*commlevel is 2 *}
      enddo
enddo
```

(a) FORTRAN D Program

```
{* Communication Generation *}
send and recv for B(i+1)
do k = 1,M
      send and recv for A(i+1)
      recv for A(i-1)
      do i = 1,N/P
            A(i) = B(i+1) + A(i+1) + A(i-1)
      enddo
      send for A(i-1)
enddo
```

(b) SPMD Program

Figure 4.10: Example of Message Generation for Loop-Carried and Loop-Independent Dependences

ative to the loop header. The compiler first inserts the primitives for independent messages; these are positioned further away from the header. The primitives for loop-carried messages are subsequently inserted preceding the loop header. This ordering avoids deadlock by allowing communication for independent messages to take place in parallel, before communicating data corresponding to carried-part messages.

Recall the program in Figure 4.4 where the messages for the *rhs* references in Sl is of type loop-carried at the k loop in the program. The boundary conditions require the compiler introduce guards to generate the communication for different RSDs for each *rhs* reference. The RSDs for the reference B(i,j+l) at the k loop level are:

Proc(l) = [2:99,3:26]
Proc(2:3)= [2:99,2:26]
Proc(4) = [2:99,2:25]

After subtracting the local index set from the above three RSDs, the compiler can determine the nonlocal index sets for the processors in each group.

The nonlocal RSDs for Proc(1:3) = [2:99,26]
Proc(4) = []

The nonlocal RSD [2:99,26] for each of processors 1-3 causes its immediate successor to send data to it. To compute the data that must be sent, the compiler translates the local indices of the receiving processors to that of the sending processors,

obtaining the section [2:99,26-25] = [2:99,1]. The message for reference B(i,j+1) is of type carried-all and is carried on loop k so the communication primitives are inserted at the beginning of the loop body. Messages for B(i,j-1) are also computed in the same way. The final result is shown in Figure 4.9.

One final aspect to consider is *storage management.* The FORTRAN D compiler provides three different storage schemes to allocate storage for nonlocal array references received from other processors. *Overlaps*, *buffers*, and *hash tables* [13] are the three options. *Overlaps* are extensions of local arrays that contain adjacent nonlocal data. They are suitable for program with high locality of references. They are permanent and specific to each array so may require more space. *Buffers* are applied in the following case where storage for nonlocal data must be reused, or the nonlocal area is bounded in size but not near the local array section. This avoids the contiguous nature of overlaps. *Hash tables* are used when the set of nonlocal elements accessed is sparse. They provide a quick look-up mechanism for arbitrary sets of nonlocal values.

To select an appropriate storage type for each nonlocal set, the compiler examines the RSDs for the nonlocal set during message generation phase. Array declarations in the generated code must be extended for nonlocal data if overlaps have been selected. If buffers are used, the new buffer declarations are inserted. All nonlocal array references in the program are modified according to their selected storage types.

4.4 Summary

Currently the FORTRAN D compiler performs message vectorization, collective communication, fine-grained pipelining, and several other optimizations for block-distributed arrays. Ongoing researches include environment support for automatic data decompositions and static performance estimation [8]. This thesis now turns its attention to a detailed investigation at the compiler's optimization techniques.

Chapter 5

FORTRAN D Compiler Optimizations

The FORTRAN D compiler uses data decomposition specifications to translate FORTRAN D programs into explicit message-passing SPMD programs to be executed on a MIMD distributed-memory machine. The compiler's goal is to generate a parallel program with low communication overhead and storage requirements. There are several advanced compiler optimizations in the existing FORTRAN D prototype compiler to accomplish this goal. Each optimization can be classified functionally into one of the following groups: (1) reducing communication startup costs, (2) hiding message copy and transmit time, (3) exploiting parallelism, or (4) reducing storage. The following lists some compiler optimizations in each group [13, 14]:

- Reducing Communication Overhead
  - Message Vectorization

  - Message Coalescing
  - Message Aggregation
  - Collective Communication
- Hiding Communication Overhead
  - Message Pipelining
  - Vector Message Pipelining
  - Iteration Reordering
  - Nonblocking Messages
- Exploiting Parallelism
  - Partitioning Computation
  - Reductions and Scans
  - Dynamic Data Decomposition
  - Pipelining Computation
- Reducing Storage
  - Partitioning Data
  - Message Blocking

5.1 Reducing Communication Overhead

Reducing communication overhead is the main goal of most parallelizing compilers. Communication overhead for each message in a parallel program is divided into three parts: T_{start}, T_{copy}, and $T_{transit}$. T_{start} is the time to setup a message; that is the startup time to send and receive messages. T_{copy} is the time to copy a message in and out of the program address space. $T_{transit}$ is the time to transfer a message between processors. Both T_{copy} and $T_{transit}$ grow with the size of a message, but

T_{start} is treated as constant.

The optimizations described here seek to reduce T_{start} by combining or eliminating messages. For most MIMD distributed-memory machines, the time to send the first byte is always much higher than the time for each additional byte. For example, in the Intel iPSC/860, the time for the first byte is approximately 240 times that of subsequent bytes [14].

5.1.1 Message Vectorization

Message vectorization significantly reduces the number of messages. It uses the result of data dependence analysis to combine several messages into a single message containing a vector of elements. This requires the compiler compute the *commlevel* for each cross-processor dependence to decide the outermost loop where element messages can be legally combined. The compiler stores each vectorized nonlocal access set at the loop level given by *commlevel*. Each vectorized nonlocal access set in a FORTRAN D compiler is represented as a RSD. Recall from chapter 4 that the code generation phase generates communication messages corresponding to these nonlocal RSDs.

5.1.2 Message Coalescing

The compiler avoids communicating redundant data by applying message coalescing to vectorized nonlocal accesses. It compares RSDs corresponding to different references on the same array and merges overlapping or contiguous RSDs. This

ensures each data value is sent to a processor only once.

Figure 5.1(a) is program has processors 1-4 own X(1:25), X(26:50), X(51:75), and X(76:100), respectively. The *owner computes* rule assigns iterations [1:25] of the k loop are executed on processor 1, iterations [26:50] are executed on processor 2, [51:75] are executed on processor 3, and [76:94] are executed on processor 4. Communication analysis shows that the references U(k+1)..., U(k+6) cause nonlocal data accesses and they are not loop-carried true dependences (Figure 5.1(b)). By applying message vectorization, the nonlocal RSDs for these references can be vectorized outside the *l* loops, resulting in the RSDs [26:26]...[26:31]. The result of message coalescing for these RSDs are [26:31]. The *send* and *recv* statements for these RSDs are inserted preceding the *l* loop (see Figure 5.2).

5.1.3 Message Aggregation

After message vectorization and coalescing, *message aggregation* is applied to the resulting RSDs to ensure that only one message is sent to each processor. It locates and aggregates all RSDs representing data being sent to the same processor. During the code generation, all the RSDs aggregated for the same processor are copied into a single buffer as one message. The receiving processor copies the received buffer to the program address space. This optimization reduces messages but requires more storage for buffering.

Consider the program in Figure 5.3(a) where the nonlocal references to $ZP(j_1\text{-}1,k_1)$ and $ZQ(j_1\text{-}1,k_1)$ inside the j_1 loop are not loop-carried true dependences. Commu-

```
REAL U(100),X(100),Y(100), Z(100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN U, X, Y, Z with D
DISTRIBUTE D(BLOCK)
do l = 1, time
      do k = 1, 94
            X(k) = F(Z(k),Y(k),U(k),...,U(k+6))
      enddo
enddo
```

(a) FORTRAN D program

when	k=20	nonlocal access set: U(26)
	k=21	nonlocal access set: U(26),U(27)
	k=22	nonlocal access set: U(26),U(27),U(28)
	k=23	nonlocal access set: U(26),U(27),U(28),U(29)
	k=24	nonlocal access set: U(26),U(27),U(28),U(29),U(30)
	k=25	nonlocal access set: U(26),U(27),U(28),U(29),U(30),U(31)

(b) The nonlocal access set for *rhs* references :U(k),U(k+1)...,U(k+6)

Figure 5.1: Example of Message Vectorization and Message Coalescing (Livermore 7-Equation of State Fragment)[14]

```
{* Compiler Output *}
REAL U(31), X(25), Y(25), Z(25)
ub1 = 25
my$proc = myproc()      {* 0..3 *}
if (my$proc .gt. 0)   send(U(1:6), my$proc-1)
if (my$proc .lt. 3)   recv(U(26:31), m$proc+1)
if (my$proc .eq. 3)   ub1 =19
do l = 1, time
      do k = 1, ub1
            X(k) = F(Z(k),Y(k),U(k),...,U(k+6))
      enddo
enddo
```

Figure 5.2: Example of Message Vectorization and Coalescing (Compiler Output)

nication from these dependences is vectorized and coalesced outside the l loop. The compiler determines after message vectorization and coalescing that these messages are sent to the same processor so it aggregates them into a single message.

The same steps are applied to nonlocal references ZR(j_2-1,k_2), ZR(j_2+1,k_2), ZQ(j_2-1,k_2), and ZQ(j_2+1,k_2) in the j_2 loop. Their references cause loop-carried true dependences so their nonlocal RSDs are vectorized and communication instructions are inserted in loop l just after the header to allow values from previous iteration to be fetched at the beginning of each new iteration. The compiler then applies message coalescing and aggregation.

Finally, the loop-independent true dependences nonlocal references to ZA(j_2-1,k_2) in the j_2 loop are considered. Its communication can be vectorized and coalescing at the level of loop l because it is the only loop common to the endpoints of the dependence. The final message to communicate will be inserted in front of the

loop k_2. The nonlocal access set for each *rhs* reference and the compiler output are shown in Figure 5.3(b) and Figure 5.4, respectively.

5.1.4 Collective Communication

Collective communication is another technique to reduce communication overhead. Instead of generating individual messages, the compiler uses fast collective communication routines, such as *broadcast*, *all-to-all*, or *transpose* in the node program. The opportunities for applying collective communication routines can be recognized by comparing the subscript expression of each distributed dimension in the *rhs* with the aligned dimension in the *lhs* reference. For example, loop-invariant (constant) subscripts in distributed array dimensions requires broadcast. Examples that use of collective communication routine are described and given in Section 5.3.2.

5.2 Hiding Communication Overhead

In this section we investigate optimizations to hide $T_{transit}$ - the message transit time - by overlapping communication with computation. The same optimizations use nonblocking messages to hide T_{copy}, the message copy time.

5.2.1 Message Pipelining

Message pipelining inserts a *send* for each nonlocal reference immediately after it is defined and the corresponding *recv* is placed immediately before the value is to be

```
REAL ZP(100,100), ZQ(100,100), ZM(100,100), S, T
REAL ZR(100,100), ZZ(100,100), ZA(100,100)
REAL ZU(100,100), ZV(100,100), ZB(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN ZA, ZB, ZM, ZP, ZQ, ZR, ZU, ZV, ZZ with D
DISTRIBUTE D(BLOCK,:)
do l = 1, time
      do k1 = 2, 99
            do j1 = 2, 99
                  ZA(j1,k1) = F1(ZP(j1-1,k1),ZQ(j1-1,k1), ZR(j1-1,k1),...)
                  ZB(j1 , k1) = F2(ZP(j1-1,k1),ZQ(j1-1,k1),...)
            enddo
      enddo
      do k2 = 2, 99
            do j2 = 2, 99
                  ZU(j2 , k2) = F3(ZZ(j2-1,k2),ZZ(j2+1,k2),ZA(j2-1,k2),...)
                  ZV(j2 , k2) = F4(ZR(j2-1,k2),ZR(j2+1,k2),ZA(j2-1,k2),...)
            enddo
      enddo
      do k3 = 2, 99
            do j3 = 2, 99
                  ZR(j3 , k3) = F5(ZR(j3,k3),ZU(j3,k3))
                  ZZ(j3 , k3) = F6(ZZ(j3,k3),ZV(j3,k3))
            enddo
      enddo
enddo
```

(a) Fortran D program

nonlocal access sets of ZP(j1-1,k1), ZQ(j1-1,k1) : ZP(0,2:99), ZQ(0,2:99), not loop-carried dependence.
ZR(j2-1,k2), ZZ(j2-1,k2) : ZR(0,2:99), ZZ(0,2:99), loop-carried ture dependence.
ZR(j2+1,k2), ZZ(j2+1,k2) : ZR(26,2:99), ZQ(26,2:99), loop-carried ture dependence.
ZA(j2-1,k2) : ZA(0,2:99), loop-independent dependence.

(b) The nonlocal access set for each *rhs* reference

Figure 5.3: Example of Message Aggregation (Livermore 18-Explicit Hydrodynamics)[14]

```
{* Compiler Output *}
REAL ZP(0:25,100), ZQ(0:25,100), ZM(0:25,100), S, T
REAL ZR(0:26,100), ZZ(0:26,100), ZA(0:25,100)
REAL ZU(25,100), ZV(25,100), ZB(25,100)
m$p =myproc()    * 0...3 *
if (m$p .lt. 3)                  send(ZP(25,2:99),ZQ(25,2:99),ZM(25,2:99),m$p+1)
if (m$p .gt. 0)                  recv(ZP(0,2:99),ZQ(0,2:99),ZM(0,2:99),m$p-1)
do l = 1, time
      if (m$p .gt. 0) send(ZR(1,2:99),ZZ(1,2:99),m$p-1)
      if (m$p .lt. 3) send(ZR(25,2:99),ZZ(25,2:99),m$p+1)
      if (m$p .lt. 3) recv(ZR(26,2:99),ZZ(26,2:99),m$p+1)
      if (m$p .gt. 0) recv(ZR(0,2:99),ZZ(0,2:99),m$p-1)
      do k1 = 2, 99
            do j1 = 1, 25
                  ZA(j1 , k1) = F1(ZP(j1-1,k1),ZQ(j1-1,k1),ZR(j1-1,k1),...)
                  ZB(j1 , k1) = F2(ZP(j1-1,k1),ZQ(j1-1,k1),...)
            enddo
      enddo
      if (m$p .lt. 3) send(ZA(25,2:99),m$p+1)
      if (m$p .gt. 0) recv(ZR(0,2:99),m$p-1)
      do k2 = 2, 99
            do j2 = 1, 25
                  ZU(j2 , k2) = F3(ZZ(j2-1,k2),ZZ(j2+1,k2),ZA(j2-1,k2),...)
                  ZV(j2 , k2) = F4(ZR(j2-1,k2),ZR(j2+1,k2),ZA(j2-1,k2),...)
            enddo
      enddo
      do k3 = 2, 99
            do j3 = 1, 25
                  ZR(j3 , k3) = F5(ZR(j3,k3),ZU(j3,k3))
                  ZZ(j3 , k3) = F6(ZZ(j3,k3),ZV(j3,k3))
            enddo
      enddo
enddo
```

Figure 5.4: Example of Message Aggregation (Compiler Output)

used. This arrangement helps to hide $T_{transit}$ by using the computation performed between the definition and use of the value. The disadvantage for this method is to prevent optimizations such as message vectorization, causing high communication cost, but it may be applied for exploiting parallelism for pipelined computation.

5.2.2 Vector Message Pipelining

Vector message pipelining uses data dependence information to place vector *send* and *recv* statements so $T_{transit}$ is hidden. Since *send* and *recv* statements interlock they must be scheduled carefully to avoid idle cycles. For example, applying vector message pipelining to invoke all *send* statements before *recv* whenever several messages are sent at the same time should improve performances.

Consider Figure 5.5(a) where the values computed by S1 are used in S3. The nonlocal references for S3 cause loop-independent true dependence from S1 to S3. The same pattern exists from S2 to S4. After message vectorization, the compiler generates the communication for S3 and S4 at the level of loop l because it is the deepest loop enclosing these true dependences. The messages for S3 and S4 are m$1 and m$2(see Figure 5.6). Vector message pipelining places the *send* for m$1 and m$2 after the j loops enclosing S1 and S2, respectively. The corresponding *recv* statements are placed before the j loops enclosing S3 and S4.

The loop-carried true dependence values computed by statements S3 (S4) are used by S1 (S2). Message vectorization creates communication for S1 and S2 at the level of l because it is the loop with the deepest loop-carried dependences. The messages

```
REAL V(1000,1000)
PARAMETER (n$proc = 10)
DECOMPOSITION D(1000,1000)
ALIGN V with D
DISTRIBUTE D(:,BLOCK)
do l = 1, time
      do j = 3, 999, 2                    {* compute red points *}
            do i = 3, 999, 2
S1:               V(i,j) = F(V(i,j-1), V(i-1,j), V(i,j+1), V(i+1,j))
            enddo
      enddo
      do j = 2, 998, 2
            do i = 2, 998, 2
S2:               V(i,j) = F(V(i,j-1), V(i-1,j), V(i,j+1), V(i+1,j))
            enddo
      enddo
      do j = 2, 998, 2                    {* compute black points *}
            do i = 3, 999, 2
S3:               V(i,j) = F(V(i,j-1), V(i-1,j), V(i,j+1), V(i+1,j))
            enddo
      enddo
      do j = 3, 999, 2
            do i = 2, 998, 2
S4:               V(i,j) = F(V(i,j-1), V(i-1,j), V(i,j+1), V(i+1,j))
            enddo
      enddo
enddo
```

(a) FORTRAN D program

nonlocal access sets of
- S1:V(i,j-1) : V(3:999:2,0), loop-carried true dependence.
- S2:V(i,j+1) : V(2:998:2,101), loop-carried true dependence.
- S3:V(i,j+1) : V(3:999:2,101), loop-independent true dependence.
- S4:V(i,j-1) : V(2:998:2,0), loop-independent true dependence.

(b) nonlocal access sets for *rhs* references.

Figure 5.5: Example of Vector Message Pipelining (Red-Black SOR) [14]

```
{* Compiler Output *}
REAL V(1000,0:101)
my$p = myproc()    * 0...9 *
if (my$p .lt. 9)   send(m$3,V(3:999:2,100),my$p+1)
do l = 1, time
      if (my$p .gt. 0) send(m$4,V(2:998:2,1),my$p-1)
      if (my$p .gt. 0) recv(m$3,V(3:999:2,0),my$p-1)
      do j = 1, 99, 2
            do i = 3, 999, 2
S1:               V(i,j) = F(V(i,j-1), V(i-1,j), V(i,j+1), V(i+1,j))
            enddo
      enddo
      if (my$p .gt. 0) send(m$1,V(3:999:2,1),my$p-1)
      if (my$p .lt. 9) recv(m$4,V(2:998:2,101),my$p+1)
      do j = 2, 100, 2
            do i = 2, 998, 2
S2:               V(i,j) = F(V(i,j-1), V(i-1,j), V(i,j+1), V(i+1,j))
            enddo
      enddo
      if (my$p .lt. 9) send(m$2,V(2:998:2,100),my$p+1)
      if (my$p .lt. 9) recv(m$1,V(3:999:2,101),my$p+1)
      do j = 2, 100, 2
            do i = 3, 999, 2
S3:               V(i,j) = F(V(i,j-1), V(i-1,j), V(i,j+1), V(i+1,j))
            enddo
      enddo
      if (my$p .lt. 9) send(m$3,V(3:999:2,100),my$p+1)
      if (my$p .gt. 0) recv(m$2,V(2:998:2,1),my$p-1)
      do j = 3, 999, 2
            do i = 2, 998, 2
S4:               V(i,j) = F(V(i,j-1), V(i-1,j), V(i,j+1), V(i+1,j))
            enddo
      enddo
enddo
if (my$p .gt. 0) recv(m$3,V(3:999:2,0),my$p-1)
```

Figure 5.6: Example of Vector Message Pipelining (Compiler Output)

for S1 and S2 are m$3 and m$4, respectively. Vector message pipelining places the *recv* for S2 just before the j loop enclosing S2 by using computation of S1 to hide $T_{transit}$.

Vector message pipelining puts the *send* for S1 after S3 and uses the computation of S4 to hide the $T_{transit}$ needed by S1. Matching *send* and *recv* statements for message m$3 must be inserted outside the *l* loop. Figure 5.5(b) and Figure 5.6 are the nonlocal access sets and the compiler output of the original program, respectively.

5.2.3 Iteration Reordering

Iteration reordering uses data dependence information to change the order of program execution. It allows loop iterations accessing local data to be separated and enclosed by a pair of *send* and *recv* statements thereby hidding $T_{transit}$.

Figure 5.7 illustrates iteration reordering. Communication analysis determines the references B(j,i-1) and B(j,i+1) access nonlocal data. The RSDs corresponding to the nonlocal accesses to array B are given by [2:99,0] and [2:99,26] (see Figure 5.7(b)). The compiler identifies and separates the iterations that access nonlocal data from iterations accessing only local data and generate a loop nest. The local iterations for the above example are given by [2:24,2:99]. Figure 5.8 is the compiler output of the original program.

```
{* FORTRAN D program *}
REAL A(100,100), B(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN A,B WITH D
DISTRIBUTE D(:,BLOCK)
do l = 1, time
      do j = 2,99
            do i = 2,99
                  A(i,j) =F(B(i,j-1),B(i-1,j),B(i+1,j),B(i,j+1))
            enddo
      enddo
      do j = 2,99
            do i = 2,99
                  B(i,j) = A(i,j)
            enddo
      enddo
enddo
```

(a) FORTRAN D program

nonlocal access sets of B(i,j-1) : B(2:99,0), loop-carried dependences.
B(i,j+1) : B(2:99,26), loop-carried dependences.

(b) The nonlocal access sets of *rhs* references

Figure 5.7: Example of Iteration Reordering

```
{*Compiler Output *}
REAL A(100,25), B(100,0:26)
my$p = myproc()    {* 0..3 *}
do l = 1, time
      if (my$p .gt. 0) send(B(2:99,1), my$p-1)
      if (my$p .lt. 3) send(B(2:99,25), my$p+1)
      {* perform local operations: only access local data *}
      do j = 2,24
            do i = 2,99
                  A(i,j) =F(B(i,j-1),B(i-1,j),B(i+1,j),B(i,j+1))
            enddo
      enddo
      if (my$p .lt. 3) recv(B(2:99,26), my$p+1)
      if (my$p .gt. 0) recv(B(2:99,0), my$p-1)
      {* perform nonlocal operations: access nonlocal data *}
      do j = 1,25,24
            do i = 2,99
                  A(i,j) =F(B(i,j-1),B(i-1,j),B(i+1,j),B(i,j+1))
            enddo
      enddo
      do j = 1,25
            do i = 2,99
                  B(i,j) = A(i,j)
            enddo
      enddo
enddo
```

Figure 5.8: Example of Iteration Reordering (Compiler Output)

5.2.4 Nonblocking Message

The FORTRAN D compiler employs blocking *send* and *recv* operations for message passing. *Send* blocks the calling processor until the data is copied from the program address space into a system buffer. A *recv* bolcks calling processor until the data has been placed into the program's address space.

Nonblocking messages permits overlapping computation and message copying. A nonblocking *send* returns control immediately to the calling processor which permits the sending processor to continue computing while data is being copied into the system buffer. Similarly, a nonblocking *recv* returns control immediately to the calling processor after posting a message destination. Therefore, the calling processor can perform computation while receiving and copying messages. The nonblocking *recv* may avoid the overhead of writing into the system buffer because the message is copied directly at the posted address. However, this requires an expensive system call to block computation until the copy is completed.

Vector message pipelining and iteration reordering with blocking messages only hides $T_{transit}$ because the processor must remain idle while copying the data. Nonblocking message provides a mechanism to hide both $T_{transit}$ and T_{copy} but it requires more systems calls. Therefore, this method should be used selectively.

5.3 Exploiting Parallelism

The goal of parallelism optimizations is to reconfigure the patterns of computation and/or communication so the amount of computation performed in parallel is minimized.

5.3.1 Partitioning Computation

The amount of parallelism extracted from a program depends on how efficiently the compiler partitions the computation. The FORTRAN D compiler uses loop bound reduction and guard introduction to achieve the partitioning.

Cross-processor dependences reduce parallelism because processors must remain idle until their predecessors complete. These dependences correspond to the sequential components of the computation that step over processor boundaries. The balance of this section describes several optimizations that extract parallelism in the presence of cross-processor dependences.

5.3.2 Reductions and Scans

Some computations allow for parallelization using *reduction* and *scan* operations in spite of cross-processor dependences. Reductions are associative and commutative operations that return a single result when applied to a collection of data. For example, a product reduction will calculate and return the product of all elements of an array. Scans are also associative and commutative operations that perform

parallel-prefix operations. For example, a product scan would compute and return the product of all the prefixes of an array. Reduction and scan operations must be associative and commutative because the compiler may change the computation order.

The FORTRAN D compiler uses dependence analysis to determine when to apply reductions and scans. The compiler can parallelize reductions or scans that contain cross-processor dependences by relaxing the *owner computes* rule and by providing methods to combine partial results. The global results can be obtained by combining the partial results either using individual *send* or *recv* calls or using *broadcast* or specialized collective communication routines such as *global_sum*(). The second method can reduce communication overhead further for common reductions. Figure 5.9 shows an example for parallelizing a sum reduction with a *global_sum* collective communication routine.

Scans are also parallelized by reordering operations. Scans allow each processor to compute its local values in parallel and communicate the partial results to all other processors. The global data is used to update local results. Figure 5.10(b) shows an example that computes a prefix sum using scan. The compiler output (Figure 5.10(a)) uses the *global_concat* communication routine to collect the partial sums from each processor in S. The partial sums collected from all the preceding processors are combined locally and used to compute local prefix sums. The final result is shown in Figure 5.10(b).

```
{* FORTRAN D Program *}
REAL X(100), Z(100), Q
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN X, Z with D
DISTRIBUTE D(block)
do l = 1, time
      Q = 0.0
      do k = 1,100
            Q = Q + Z(k) * X(k)
      enddo
enddo
{* Compile Output *}
REAL X(25), Z(25), Q
do l = 1, time
      Q = 0.0
      do k = 1,25
            Q = Q + Z(k) * X(k)
      enddo
      Q = global_sum()      {* sum reduction function *}
enddo
```

Figure 5.9: Example of Reduction (Inner Product) [14]

```
{* FORTRAN D Program *}
REAL X(100), Y(100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN X, Y with D
do l = 1, time
    X(1) = Y(1)
    do k = 2,100
        X(k) = X(k-1) + Y(k)
    enddo
enddo
```

(a) FORTRAN D Program

```
{* Compile Output *}
REAL X(25), Y(25), S(0:3)
my$p = myproc()    {* 0...3 *}
do l = 1, time
    S(my$p) = 0.0
    do k = 1,25
        S(my$p) = S(my$p) + Y(k)
    enddo
    global_concat(S)
    X(1) = Y(1)
    if (my$p .ne. 0) then
        do k = 0, my$p-1
            X(1) = X(1) + S(k)
        enddo
    endif
    do k = 2,25
        X(k) = X(k-1) + Y(k)
    enddo
enddo
```

(b) Compiler Output

Figure 5.10: Example of Scan (First Sum)

5.3.3 Dynamic Data Decomposition

In certain cases, the *owner computes* rule partitions an otherwise parallel computation in a way that causes sequential execution. *Dynamic data decomposition* can be used to temporarily change data ownership when this occurs. This localizes cross-processor dependences and achieves the desired parallelism. The decomposition method is only applicable when there are full dimensions of parallelism available in the computation.

Figure 5.11(a) illustrates a computation *wavefront* that crosses a spatial dimension in each phase. A fixed column or row data distribution is not suitable to both phases. The FORTRAN D compiler applies dynamic data decomposition to parallelize the sequential phase using collective communication routines to change the array decomposition after each phase. After changing the data decomposition, the computation wavefront in both phases are internalized and all the processors can be executed in parallel without communication (see Figure 5.11(b)).

5.3.4 Pipelining Computation

Pipelining can often be used to extract partial parallelism from computations that contain cross-processor dependences. In parallel computations all processors may execute concurrently and communicate data as necessary but a pipeline means a processor can execute only after it receives results computed by its predecessor. Pipelining allows processors to overlap their computations by sending the partial results to their successors earlier. The degree of pipeline parallelism depends on how soon each processor is able to start working after its predecessor begins. Fig-

```
REAL A(100), B(100), X(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN X WITH D
DISTRIBUTE D(:,BLOCK)
do l = 1, time
    do j = 1, 100            {* Phase 1: sweep along columns *}
        do i = 2, 100
            X(i,j) =F1(X(i,j),X(i-1,j),A(i),B(i))
        enddo
    enddo
    do j = 2, 100            {* Phase 2: sweep along rows *}
        do i = 1, 100
            X(i,j) =F2(X(i,j),X(i,j-1),A(i),B(i))
        enddo
    enddo
enddo
```

(a) FORTRAN D Program

```
REAL A(100), B(100), X(100,25), X1(25,100)
EQUIVALENCE (X,X1)
do l = 1, time
    do j = 1, 25             {* Phase 1: sweep along columns *}
        do i = 2, 100
            X(i,j) =F1(X(i,j),X(i-1,j),A(i),B(i))
        enddo
    enddo
    redistribute_row_to_col(X)      {* Dynamic Data Decomposition *}
    do j = 2, 100            {* Phase 2: sweep along rows *}
        do i = 1, 25
            X(i,j) =F2(X(i,j),X(i,j-1),A(i),B(i))
        enddo
    enddo
    redistribute_col_to_row(X1)     {* Dynamic Data Decomposition.*}
enddo
```

(b) Compiler Output

Figure 5.11: Example of Dynamic Data Decomposition (ADI Integration) [13]

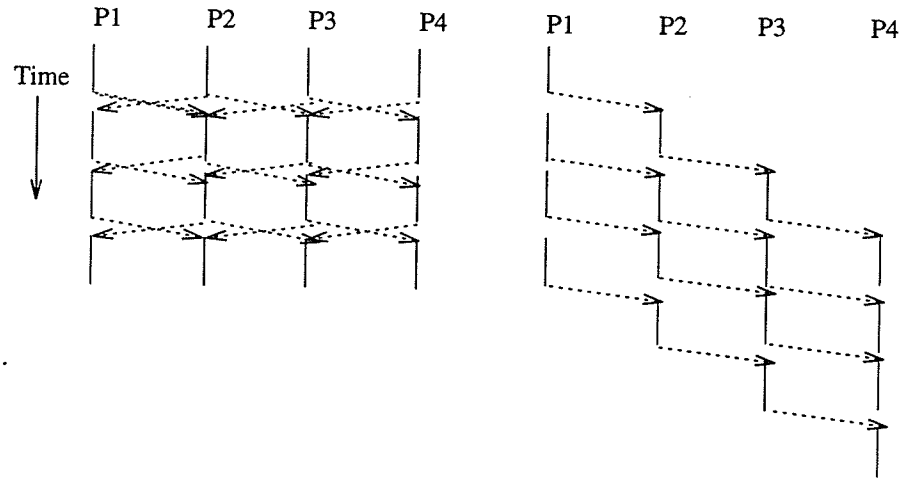


Figure 5.12: Parallel and Pipelined Computation

ure 5.12 illustrates time-space diagrams of pipelined and parallel computations.

The FORTRAN D compiler uses cross-processor loops to distinguish between pipelined and fully parallel computation. The presence of any cross-processor loop in a loop nest indicates that it is a pipelined computation. Cross processor loop causes computation wavefronts to sweep across processor boundaries. The Fortran D compiler uses an algorithm that considers all pairs of array references that cause loop-carried true dependences. If the subscript expressions in an array's distributed dimension are not identical all loop index variables belong to cross-processor loops.

The granularity of pipelined computation is determined by the amount of computation enclosed by cross-processor loops. Under fine-grain pipelining all cross-processor loops are rearranged as deeply as possible to minimize the amount of computation. The resulting program would achieve the finest pipelining granularity by generating messages needed by other processors in the shortest time. This

```
{* FORTRAN D program *}
REAL ZA(100,100), ZB(100,100), ZR(100,100), QA
REAL ZU(100,100), ZV(100,100), ZZ(100,100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100,100)
ALIGN ZA, ZB, ZR, ZU, ZV, ZZ WITH D
DISTRIBUTE D(:,BLOCK)
do l = 1, time
    do j = 2,99
        do k = 2,99
S1:         QA =F1(ZA(k,j+1),ZA(k,j-1),ZA(k+1,j),ZA(k-1,j))
S2:         ZA(k,j) =F2(ZA(k,j),QA)
        enddo
    enddo
enddo
```

Figure 5.13: Example of Pipelining Computation (Implicit Hydrodynamics)

results in high message overhead because a message is sent for each iteration accessing nonlocal data.

Coarse-grain pipelining increases the amount of computation (C) enclosed by cross-processor loops by applying *loop interchanges* and *strip-mining*. Under coarse-grain pipelining, communication can be reduced by increasing C but this will weaken parallelism because processors must wait longer before starting computation.

The program segment in Figure 5.13 contains a loop-carried true dependence between S1 and S2 due to the references ZA(k,j) and ZA(k,j-1). The compiler determines the second dimension of ZA is distributed and subsequently compares the subscripts j and j-1. The j loop is labeled as a cross-processor loop, because the subscripts are unequal.

```
{* Compiler Output1: Fine-graining pipelining *}
REAL ZA(100,0:26), ZB(100,25), ZR(100,25), QA
REAL ZU(100,25), ZV(100,25), ZZ(100,25)
my$p = myproc() {* 0..3 *}
do l = 1, time
    if (my$p .gt. 0) send(ZA(0:99,1),my$p-1)
    if (my$p .lt. 3) recv(ZA(0:99,26),my$p+1)
    do k = 2,99
        if (my$p .gt. 0) recv(ZA(k,0),my$p-1)
        do j = 1,25
            QA =F1(ZA(k,j+1),ZA(k,j-1),ZA(k+1,j),ZA(k-1,j))
            ZA(k,j) =F2(ZA(k,j),QA)
        enddo
    enddo
        if (my$p .lt. 3) send(ZA(k,25),my$p+1)
enddo
```

(a) Compiler Output$_1$

```
{* Compiler Output2: Course-graining pipelining *}
REAL ZA(100,0:26), ZB(100,25), ZR(100,25), QA
REAL ZU(100,25), ZV(100,25), ZZ(100,25)
my$p = myproc() {* 0..3 *}
do l = 1, time
    if (my$p .gt. 0) send(ZA(0:99,1),my$p-1)
    if (my$p .lt. 3) recv(ZA(0:99,26),my$p+1)
    do kk = 2,99, B      {* Strip-mining, B is the size of the strip-mining *}
        if (my$p .gt. 0) recv(ZA(kk:kk+B-1,0),my$p-1)
        do j = 1,25
            do k = kk,kk+B
                QA =F1(ZA(k,j+1),ZA(k,j-1),ZA(k+1,j),ZA(k-1,j))
                ZA(i,j) =F2(ZA(i,j),QA)
            enddo
        enddo
    enddo
    if (my$p .lt. 3) send(ZA(kk:kk+B-1,25),my$p+1)
enddo
```

(b) Compiler Output$_2$

Figure 5.14: Example of Pipelining Computation (Implicit Hydrodynamics)

Compiler-output$_1$ in Figure 5.14(a) illustrates fine-grain pipelining. The compiler repositions the cross-processor loop j to the innermost position to maximize pipelining. Compiler-output$_2$ in Figure 5.14(b) illustrates course-grain pipelining. It strip-mines the k loop by a factor B, then interchanges the iterator loop kk outside the j loop. The compiler vectorizes the communication for B iterations at the j loop.

5.4 Reducing Storage

Usually program optimizations require more temporary storage. Compile-time storage optimization is fundamental to an efficient compiler, so partitioning data so that each processor allocates memory only for its local data is useful. The FORTRAN D compiler provides three types of temporary storage schemes for nonlocal data: overlaps, buffers, and hash table. If insufficient storage is available, message blocking strip-mines loops by a block factor (B). Each vectorized message of size n is then divided into n/B messages of size B. This reduces the buffer space required by a factor of n/B at the expense of additional messages.

5.5 Optimization Algorithm

Figure 5.15 provides a high level description of the overall FORTRAN D compiler optimization algorithm. It describes how optimizations are organized in the FORTRAN D compiler.

```
partition data across processor
partition computation using owner computes rule
detect and parallelize reductions and scans
compute cross-processor loops
for each loop nest L do
    if L is fully parallel (i.e., no cross-processor loops) then
        vectorize, coalesce, and aggregate messages
        select and insert collective communications
        if sufficient T_comp exists to hide T_copy, T_transit then
            apply vector message pipelining
            insert nonblocking messages
        else if T'comp can be profitably created and used then
                reorder iterations
                apply vector message pipelining
                insert nonblocking messages
            else insert blocking messages
        endif
    else  {* must be pipelined computation *}
        select efficient granularity for pipelining
        apply strip-mining and loop interchange
        vectorize and coalesce, and aggregate messages
        insert blocking messages
    endif
    if insufficient storage is available then
        apply storage optimizations
    endif
enddo
```

Figure 5.15: Compiler Optimization Algorithm

5.6 Program Transformations

The FORTRAN D compiler also applies *program transformations* to expose or enhance parallelism in the source code. Program transformations are useful for shared-memory parallelizing compilers. They use dependence information to determine their *legality* and *profitability*. To determine the legality of each transformation in distributed-memory compilers is the same for the shared-memory compilers. Their profitability criteria expose and enhance parallelism in the source code and reduce the communication overhead in the SPMD code. Each of the following must be considered.

Loop Distribution: separates the statements in a single loop into multiple loops with identical lower bounds, upper bounds, and steps. Figure 5.16(b) shows the result of applying loop distribution to the loop in Figure 5.16(a).

Loop Fusion: combines multiple loops with same lower bounds, upper bounds, and steps into a single loop. Figure 5.17(b) shows the fusion result of the loops in Figure 5.17(a).

Loop Interchange: changes the traversal order of adjacent loop headers. Loop interchange can be used to increase the chance of applying message aggregation in FORTRAN D program. Figure 5.18(b) is the result after applying loop interchange to the loops in Figure 5.18(a).

Strip Mining: increases the step size of an existing loop and adds an additional inner loop. This method can break large messages into small packets in the SPMD codes. Figure 5.19(b) is the result after applying strip mining to the loops in Figure 5.19(a).

```
      for i = 1 to N do
S1:       ...
S2:       ...
      endfor
```

(a) Before Loop Distribution

```
      for i = 1 to N do
S1:       ...
      endfor
      for i = 1 to N do
S2:       ...
      endfor
```

(b) After Loop Distribution

Figure 5.16: Loop Distribution

```
      for i = 1 to N do
S1:       ...
      endfor
      for i = 1 to N do
S2:       ...
      endfor
```

(a) Before Loop Fusion

```
      for i = 1 to N do
S1:       ...
S2:       ...
      endfor
```

(b) After Loop Fusion

Figure 5.17: Loop Fusion

```
        for i = 1 to N do
            for j = 1 to N do
S1:             ...
S2:             ...
            endfor
        endfor
```

(a) Before Loop Interchange

```
        for j = 1 to N do
            for i = 1 to N do
S1:             ...
S2:             ...
            endfor
        endfor
```

(b) After Loop Interchange

Figure 5.18: Loop Interchange

```
        SEND(B(1:n))
        RECEIVE(B(1:n))
        for i = 1 to n do
S1:         =B(i)...
        endfor
```

(a) Before Strip Mining

```
        for i = 1 to N ,b do
            SEND(B(i:i+b-1))
            RECEIVE(B(i:i+b-1))
            for j = i to i+b-1 do
S1:             =B(j)...
            endfor
        endfor
```

(b) After Strip Mining

Figure 5.19: Strip Mining

Testing the legality of these optimizations is identical to that of shared-memory compilers.

5.7 Summary

The existing FORTRAN D compiler prototype parallelizes reductions, pipelined computations, and performs message vectorization, coalescing, and aggregation for block-distributed arrays. The remaining optimizations are still under implementation and their effectiveness are under evaluation for larger and more varied programs on different MIMD architectures, including networks of workstations.

Chapter 6

Compiler Optimization for Parallel Computation

Chapter 5 discusses the (proposed) compiler optimizations in the FORTRAN D compiler. All these optimizations yield significant improvement by reducing communication overhead. During the optimization process, the compiler determines the communication between any two processors and restructures the code to reduce message traffic between them. This chapter describes a new compiler optimization that reduces message traffic by analyzing communication patterns among all the processors.

Dependences constrain parallel execution of programs. The dependence structure of a computation can be modeled using a directed acyclic graph (DAG) where the nodes represent tasks and edges represent precedence constraints. Dependence edges due to flow dependences define message-passing synchronization in a distributed-memory machine model of parallel execution. If a dependence edge

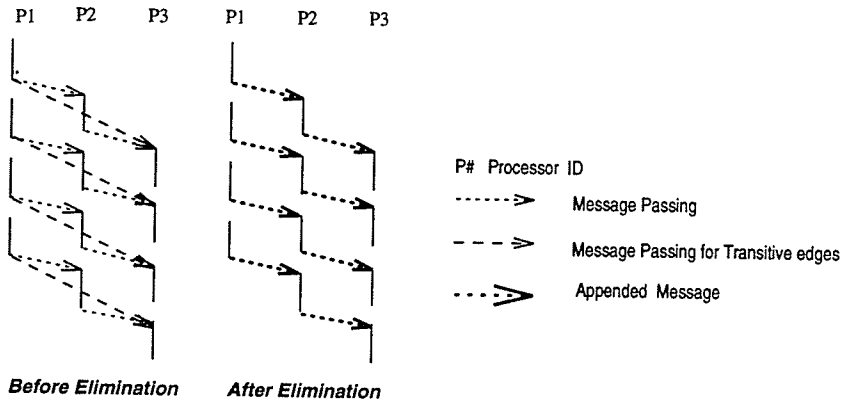


Figure 6.1: Eliminating Messages for Pipelined Computation with Transitive-Edge Dependences

in the DAG is transitive there is an alternative path composed of edges connecting the source and sink of that dependence.

The startup cost of a message transmission is much higher than the actual data transfer cost. Thus, if several messages can be grouped and sent as a single message communication overhead can be significantly reduced. Message transfers for dependences corresponding to transitive edges can be rerouted, without losing parallelism, by appending the data to the messages on the alternative path that covers the transitive edge (see Figure 6.1).

This chapter describes a new compiler-optimization technique for pipelined computation. The method identifies the dependences corresponding to transitive edges in loops with constant dependences and introduces a scheme for assigning storage for nonlocal data corresponding to the transitive edges in the node program. This technique can be applied to single and double nested loops. The dependences discussed here are limited to flow dependences.

6.1 Iteration Space Dependence Graph

Parallel execution of loop iterations is constrained by dependences. The inter-iteration dependences of a loop can be formally represented by an *Iteration Space Dependence Graph* (ISDG). An ISDG is a directed acyclic graph G(V,E) where V and E are the set of nodes and the set of edges, respectively. Each node in the graph represents one iteration in the loop. An edge (V_i ,V_j) in E signifies the existence of a dependence from the iteration denoted by V_i to the iteration denoted by V_j. The dependence distance of each edge is given by subtracting the iteration number of the source of a dependence from that of its sink. An ISDG is classified as regular if the existence of a dependence with distance d_i from a node implies the existence of such a dependence from each node that has an adjacent point at a distance d_i in the iteration space. A regular loop is an iteration loop with a regular ISDG.

These concepts are illustrated by the program in Figure 6.2(a). In this FORTRAN D program, array X is cyclically distributed among 4 processors. Figure 6.2(b) shows the distribution pattern of array X on each processor after compilation. The ISDG of the regular loop is shown in Figure 6.2(c). In this ISDG, each node represents one loop iteration. Any edge drawn between two nodes signifies the existence of a dependence. The flow dependence from iteration *i* to iteration *i+1* manifests itself as an edge from node *i* to node *i+1* in the ISDG. The same applies for the dependence from iteration *i* to iteration *i+2*. Figure 6.3 shows the SPMD code generated using the conventional optimization techniques described in chapter 5. The compiler generates a pair of *send* and *recv* messages for each dependence in

the SPMD code.

These dependence edges correspond to synchronization events of parallel computation which in a MIMD distributed-memory machines is achieved through message-passing. In general, the sparser the ISDG, the smaller the communication overhead is in a parallel computation. Any of the transitive edges may be removed from the ISDG by rerouting the corresponding message through its alternative path. A dependence-edge from iteration i to iteration k is a transitive edge if and only if :

1. there exists a sequence of iterations i_1, i_2,$\cdots$,i_j such that j>0 and there is a dependence from i_m to i_{m+1} for $1 \leq m < j$, and
2. there is a dependence from iteration i to iteration i_1 and from iteration i_j to iteration k.

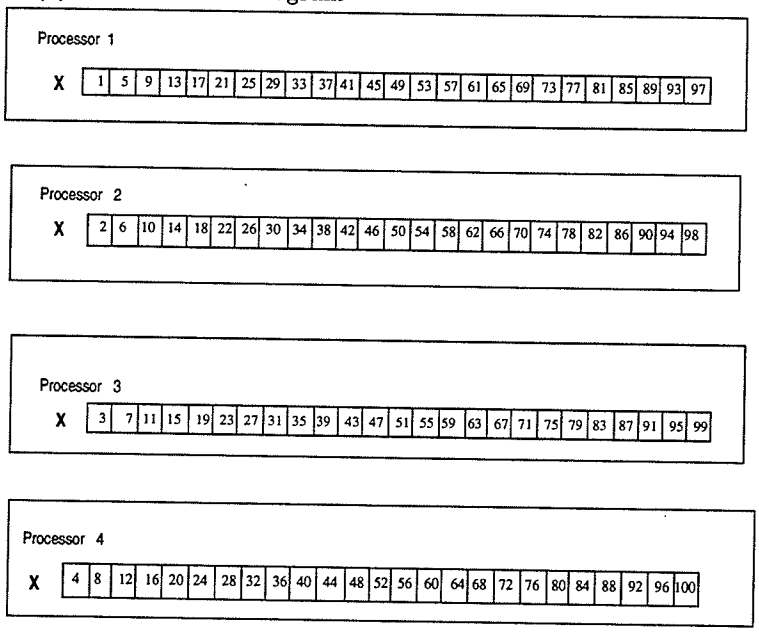
Elimination of transitive-edge dependences reduces message traffic in the parallel execution of a loop. In the next section, we present a technique to identify the dependences corresponding to a transitive edge and eliminate the message due to this edge by re-routing it through its alternative path.

6.2 Identifying Transitive-Edge Dependences

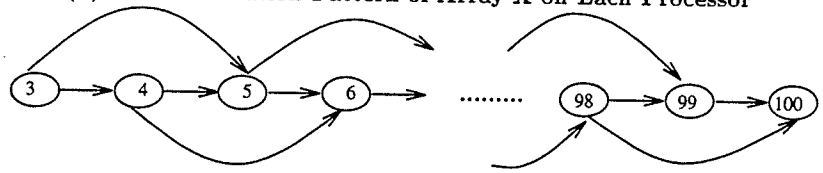
To identify transitive-edge dependences for a regular loop, the first step is to build the ISDG of the loop. When an ISDG is regular, its transitive edges can be identified by analyzing a subgraph of the ISDG. This subgraph, called R_subgraph, is the

```
REAL X(100)
PARAMETER (n$proc = 4) {* the number of processors *}
DECOMPOSITION D(100)
ALIGN X WITH D
DISTRIBUTE D(CYCLIC)
      do k= 3, 100
            X(k) = X(k-1)+X(k-2)
      enddo
```

(a) FORTRAN D Program



(b) The Distribution Pattern of Array X on Each Processor



(c) ISDG of the loop

Figure 6.2: Example of ISDG

```
{* compiler output *}
REAL X(25), temp1, temp2
my$proc= myproc() {* 0···3 : the processor ID of each processor *}
{* next : the succeeding processor whose PID is ((my$proc+1) mod n$proc) *}
{* nnext : the processor succeeding next whose PID is ((my$proc+2) mod n$proc) *}
{* prev : the preceding processor whose PID is ((my$proc-1+n$proc) mod n$proc) *}
{* pprev : the processor preceding prev whose PID is ((my$proc-2+n$proc) mod n$proc) *}

      lb1=1
      ub1=25
      if (my$proc < 2) {* boundary condition *}
            lb1 = 2
            send(X(1),next)
            send(X(1),nnext)
      endif
      do k= lb1, ub1
            recv(temp1,prev)
            recv(temp2,pprev)
            X(k)=temp1+temp2
            send(X(k),next)
            send(X(k),nnext)
      enddo
```

Figure 6.3: SPMD Program

induced subgraph of an ISDG. The size of the subgraph is independent of the size of the iteration space but depends on the values of dependence distances [12]. The next section describes how to build the R_subgraph and identifies transitive-edges in this subgraph for a non-nested or a double-nested loop.

6.2.1 Regular Non-Nested Loops

For a regular nonnested loop the lower bound of the loop control variable is denoted *lower1* and without loss of generality there are m dependences. Let D = { d_1, d_2,$\cdots$, d_m } be the set of dependence distances and dmax = MAX(D) be the largest value of the set D. A dependence distance is given subtracting the iteration number of a dependence's source from its sink so any d_i ($1 \leq i \leq m$) is positive. The vertex set (V$\prime$) and edge set (E$\prime$) of the R_subgraph for this non-nested loop is defined as follows:

V$\prime$= { *lower1*, *lower1*+1 ,$\cdots$, dmax+*lower1* }
E$\prime$= {(i,j) | iff there exists a d_k such that j=i+d_k,
for every *lower1* $\leq$ i < j $\leq$ (dmax+*lower1*)}

The R_subgraph of the ISDG in Figure 6.2(c) is given in Figure 6.4(a). In the program of Figure 6.2(a), the lower bound of the loop control variable is 3 and there are two flow dependences in this loop: the set of dependence distances (D) is {1,2} and dmax is 2. The V$\prime$ and E$\prime$ of the R_subgraph are:

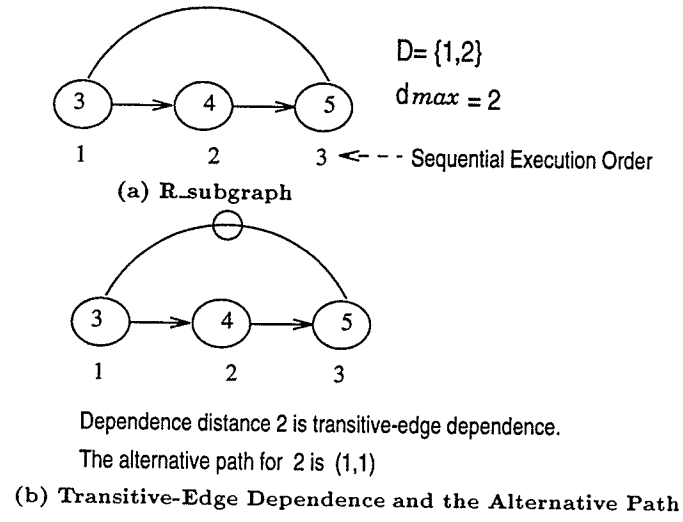


Figure 6.4: Example of R_subgraph and Transitive-Edge Dependences

$V\prime = \{ 3, 4, 3+2 \}$

$E\prime = \{ (3,4), (4,5), (3,5) \}$

Determining the transitive edges in an ISDG only requires the determination of the transitive edges at node *lower1* [12]. The algorithm to identify the transitive edges, a variant of Depth First Search (DFS), is given in Figure 6.5 and 6.6.

This algorithm identifies the transitive-edge dependences from the node *lower1* of the R_subgraph. Let $V\prime\prime$ be the set of all vertices reachable from the node *lower1* of the R_subgraph and $G(V\prime\prime,E\prime\prime)$ be the induced subgraph of R_subgraph on $V\prime\prime$. The definition of $V\prime\prime$ implies that for any node i, there exists a path from node *lower1* to node i in G. The algorithm given in Figure 6.5 is applied to find a spanning tree

```
DO i := 1 to N {* N is the number of vertex set *}
    node[i].visited := false;
END
Forward_Set = {};
count := 0;
v := 1;
node[v].parent = v;
WHILE (count ≠ N) DO
BEGIN
    IF node[v].visited = false THEN
    BEGIN
        count := count +1;
        node[v].visited := true;
        Let S be the set of unvisited nodes in the adjacency list of v
        IF |S| ≠ 0 THEN
        BEGIN
            for all but the smallest u in S push (v,u)
            onto the stack in decreasing SEO (Sequential Execution Order) of u.
            v/ = v;
            set v to the smallest element of S;
            node[v].parent = v/;
        END
    END
    ELSE IF stack is not empty THEN
        REPEAT
            pop(stack(u,v));
            IF node[v].visited = true THEN
                Forward_Set = Forward_Set + {(u,v)};
            ELSE
                node[v].parent = u;
        UNTIL node[v].visited is false or stack is empty;
END;
```

Figure 6.5: Sequential Execution Order Depth First Search Algorithm

```
FOR each (u,v) in Forward_Set DO
BEGIN
    push((node[v].parent,v));
    v/ = node[v].parent;
    WHILE (v/ ≠ u) DO
    BEGIN
        push(node[v/.parent,v/);
        v/ = node[v/].parent;
    END
    PRINT("Replaced path for forward edge",(u ,v ),"  is");
    REPEAT
        pop(stack(node[v/].parent,v/));
        PRINT((node[v/.parent,v/));
    UNTIL stack is empty;
END
```

Figure 6.6: Sequential Execution Order Depth First Search Algorithm(Cont.)

(T(V//,E///)) of G, rooted at the node *lower1*. An edge (i,j) of G(V//,E//) can be classified as: (1) tree edge, if it belongs to T, (2) forward edge, if it does not belong to T and there exists a path from i to j in T, or (3) cross edge, if it does not belong to T and there is no path from i to j in T.

A forward edge of G(V//,E//) with respect to a DFS spanning tree is a transitive edge of R_subgraph [12]. The alternative path for each transitive edge dependence can be found by using the alternative path for each forward edge in the spanning tree T. Figure 6.4(b) shows the transitive-edge and its alternative path found in the R_subgraph of Figure 6.4(a). Dependence distance 2 is identified as a transitive-edge dependence and its alternative path is (1,1).

6.2.2 Regular Double-Nested Loops

This section considers the double-nested loops that have a rectangular iteration space where the lower and upper bound of the inner loop are independent of the iteration number of the outer loop. Unlike simple loops, the presence of a mix of negative and positive values in the second components of the dependence vectors increases the complexity required to construct the R_subgraph for rectangular loops. This section describes the construction of the R_subgraph for double nested loop.

The lower bounds of the loop control variables are *lower1* and *lower2*. The dependence distances of two-dimensional loops are vectors. The first component of a dependence vector corresponds to the outer loop and the second to the inner. The component corresponding to the inner loop of a dependence distance vector can be negative; that is, the iteration number corresponding to the source of a dependence can be higher than the sink of the dependence. In any case, the first non zero component of a dependence distance vector is positive. Here we denote a dependence distance vector d_i by its component (d_{i1}, d_{i2}). Let $D = \{d_1, d_2, \cdots, d_m\}$ be the set of dependence distance vectors in a rectangular iteration space. Depending on the sign of the second components, D can be classified as non-negative, non-positive, or mixed. The following examples illustrate a method to construct the R_subgraph for each of these cases.

Case One: *the second component of all the dependence vectors are non-negative.* Let $dmax1 = \text{MAX}(\{ d_{i1} \mid (d_{i1}, d_{i2}) \in \text{to } D\})$, $dmax2 = \text{MAX}(\{ d_{i2} \mid (d_{i1}, d_{i2}) \in \text{to } D \})$, and $d_{i2} \geq 0$, for all $(d_{i1}, d_{i2}) \in$ to D. The R_subgraph $(R(V\prime, E\prime))$ of

the double-nested loop is defined by the rectangular iteration space bounded by (*lower1*, *lower2*) and (d*max1*+ *lower1*, d*max2*+*lower2*).

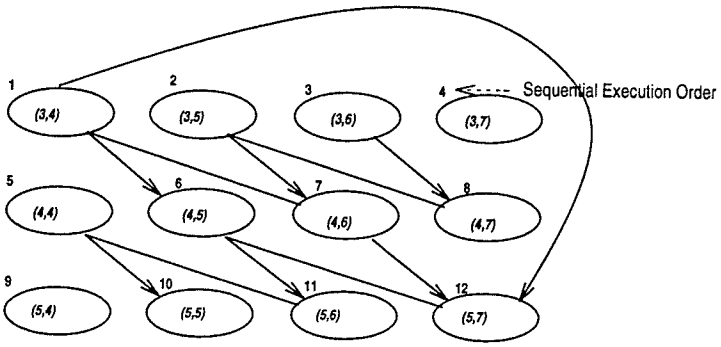
$V\prime = \{(lower1, lower2),(lower1,lower2+1), \cdots,(lower1, dmax2 + lower2),\cdots,(dmax1 + lower1, dmax2 + lower2)\}$
$E\prime = \{((i_1,j_1),(i_2,j_2))$ | iff there exists a dependence vector d_k in D such that $(i_2,j_2) = (i_1,j_1) + d_k$, for every $(lower1, lower2) \leq (i_1, j_1) < (i_2, j_2) \leq (dmax1 + lower1, dmax2 + lower2)$ }

These concepts are illustrated using the example in Figure 6.7. Figure 6.7(a) The lower bounds of loop control variables (*lower1*, *lower2*) are (3,4) (Figure 6.7(a)). The dependence distance set D for this loop is {(1,1),(1,2),(2,3)}. The R_subgraph for this loop is given in Figure 6.7(b). The vertices of this graph are labeled according to their sequential execution order and it is sufficient to determine the transitive edges at node (*lower1*, *lower2*). The algorithm in Figure 6.5 is used to identify the transitive edges and the corresponding alternative path. Figure 6.7(c) illustrates the transitive edges in the R_subgraph of Figure 6.7(b). Dependence distance vector (2,3) is identified as a transitive edge dependence and its alternative path is ((1,1),(1,2)).

Case Two: *the second component of all the dependence vectors is non-positive.* Let $dmax1 = MAX(\{ d_{i1} \mid (d_{i1},d_{i2}) \in \text{to D} \})$, $dmax2 = MAX(\{ abs(d_{i2}) \mid (d_{i1},d_{i2}) \in \text{to D}\})$ and $d_{i2} \leq 0$, for all $(d_{i1},d_{i2}) \in$ to D. The R_subgraph (R(V/, E/)) of the double-nested loop can be defined by the rectangular iteration space bounded by (*lower1* ,*lower2*) and (d*max1*+ *lower1*, d*max2*+*lower2*).

```
A(100,100),B(100,100)
Parameter (n$proc = 100)
Decomposition D(100,100)
Align A,B with D
Distribute D(CYCLIC,BLOCK)
do i = 3 , 100
    do j = 4 , 100
        A(i,j) = B(i-1,j-1)
        B(i,j) = B(i-1,j-2) + A(i-2,j-3)
    end
end
```

(a) FORTRAN D Program

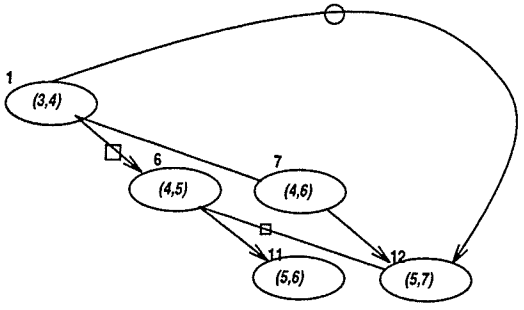


D = {(1,1), (1,2), (2,3)}

$d_{max1} = 2$

$d_{max2} = 3$

(b) R_subgraph



Dependence distance vector (2,3) is transitive-edge dependence.

The alternative path for (2,3) is ((1,1),(1,2))

(c) Transitive-Edge Dependence and the Alternative Path

Figure 6.7: Case One Illustrated

$V\prime = \{ (lower1,lower2),(lower1,lower2+1), \cdots,(lower1, dmax2+lower2), \cdots,(dmax1 + lower1, dmax2 + lower2)\}$

$E\prime = \{((i_1,j_1),(i_2,j_2))$ | iff there exists a dependence vector d_k in D such that $(i_2,j_2) = (i_1,j_1) + d_k$, for every $(lower1, lower2) \leq (i_1, j_1)$, $(i_2, j_2) \leq (dmax1 + lower1, dmax2 + lower2)$ }

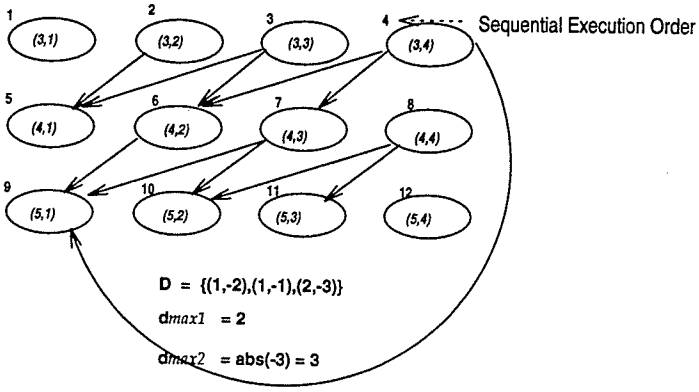
Figure 6.8 illustrates these concepts. In program of Figure 6.8(a), the lower bounds of loop control variables (*lower1, lower2*) are (3,1). The dependence distance set D for this loop is {(1,-1),(1,-2),(2,-3)}. The R_subgraph for this loop is given in Figure 6.8(b). The vertices of this graph are numbered according to their sequential execution order and it is sufficient to determine the transitive edges at node (*lower1,lower2*+d*max2*). The algorithm in Figure 6.5 is used to identify the transitive edges and the corresponding alternative paths. Figure 6.8(c) illustrates the transitive edges in the R_subgraph of Figure 6.8(b). Dependence distance vector (2,-3) is identified as a transitive-edge dependence and its alternative path is ((1,-2),(1,-1)).

Case Three : *the second component of the dependence vectors consists of a mix of positive and negative values.* Let $dmax1 = MAX(\{ d_{i1} \mid (d_{i1},d_{i2}) \in$ to D$\})$, $pmax = MAX(MAX(\{ d_{i2} \mid (d_{i1},d_{i2}) \in$ to D$\}), 0)$, $pmin = MIN(MIN(\{ d_{i2} \mid (d_{i1},d_{i2}) \in$ to D$\}), 0)$ In other words, $pmax$ is the largest positive value and $pmin$ is the smallest negative value.

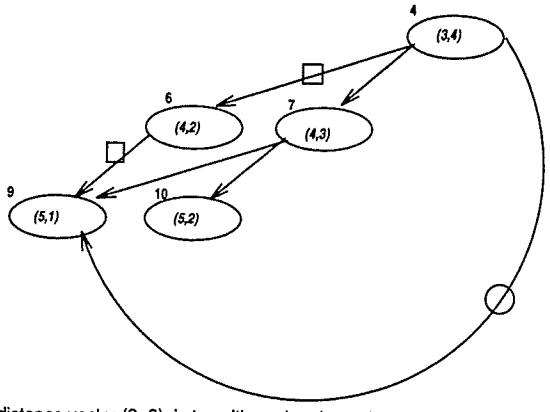
The R_subgraph, R(V$\prime$, E$\prime$) of the double-nested loop can be defined by the rectangular iteration space bounded by (*lower1, lower2*) and ($dmax1 + lower1$, $pmax$

```
A(100,100),B(100,100)
Parameter (n$proc = 100)
Decomposition D(100,100)
Align A,B with D
Distribute D(BLOCK,CYCLIC)
do i = 3 , 100
        do j = 1 , 100
            A(i,j) = B(i-1,j+2)
            B(i,j) = B(i-1,j+1) + A(i-2,j+3)
        end
end
```

(a) FORTRAN D Program



(b) R_subgraph



Dependence distance vector (2,-3) is transitive-edge dependence.
The alternative path for (2,-3) is ((1,-2),(1,-1))

(c) Transitive-Edge Dependence and the Alternative Path

Figure 6.8: Case Two Illustrated

+ abs(p*min*) + *lower2*) where *lower1* and *lower2* are the lower bounds of the outer and inner loops, respectively.

V$'$ = { (*lower1*,*lower2*),(*lower1*,*lower2* + 1), $\cdots$,(*lower1*, p*max* + abs(p*min*) + *lower2*), $\cdots$,(d*max1* + *lower1*, p*max* + abs(p*min*) + *lower2*)}

E$'$ = {((i$_1$,j$_1$),(i$_2$,j$_2$)) | iff there exists a dependence vector d$_k$ in D such that (i$_2$,j$_2$) = (i$_1$,j$_1$) + d$_k$, for every (*lower1*, *lower2*) $\leq$ (i$_1$, j$_1$), (i$_2$, j$_2$) $\leq$ (d*max1* + *lower1*, p*max* + abs(p*min*) + *lower2*) }

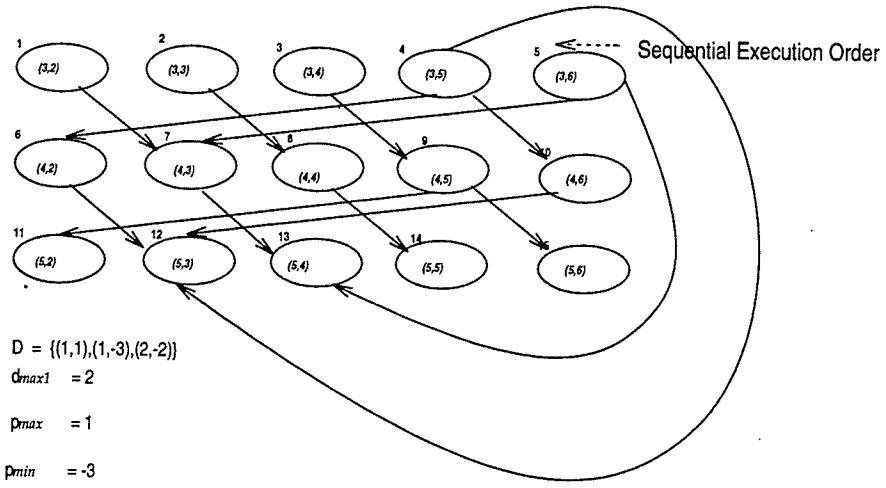
This case is illustrated in Figure 6.9. In the program of Figure 6.9(a), the lower bounds of the loop control variables (*lower1*,*lower2*) are (3,2). The dependence set D for this loop is {(1,1),(1,-3),(2,-2)}. The R_subgraph for this loop is illustrated in Figure 6.9(b). The vertices of this subgraph is labeled according to their sequential execution order and it is sufficient to identify transitive edges at node (*lower1*, *lower2*+abs(p*min*)). The algorithm in Figure 6.5 is used to identify the transitive edges and the corresponding alternative paths. Figure 6.9(c) illustrates the transitive edges in the R_subgraph of Figure 6.9(b). Dependence distance vector (2,-2) is identified as a transitive-edge dependence and its alternative path is ((1,-3),(1,1)).

6.3 Implementation on the FORTRAN D Compiler

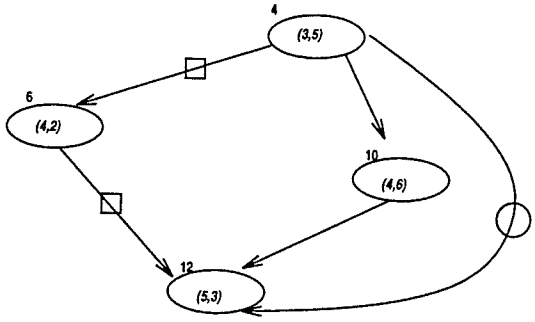
The compiler identifies the transitive edges in the program analysis phase. Rerouting message corresponding to a transitive edge could be accomplished either by

```
A(100,100),B(100,100)
Parameter (n$proc = 100)
Decomposition D(100,100)
Align A,B with D
Distribute D(cyclic,cyclic)
do i = 3 , 100
    do j = 2 , 97
        A(i,j) = B(i-1,j-1)
        B(i,j) = B(i-1,j+3) + A(i-2,j+2)
    end
end
```

(a) FORTRAN D Program



(b) R_subgraph



Dependence distance vector (2,-2) is transitive-edge dependence.

The alternative path for (2,-2) is ((1,-3)(1,1))

(c) Transitive-Edge Dependence and the Alternative Path

Figure 6.9: Case Three Illustrated

changing the communication analysis algorithm or by means of source to source transformation of the loop. Both the schemes allocate storage for rerouted message in the intermediate processors and rely on "message aggregation" optimization to combine the rerouted message with another message. This section details of the methodology.

6.3.1 Modifying Communication Analysis

During the communication analysis phase the compiler generates **IN** and **OUT** sets of a reference for each processor pair. The set IN_{ij} of a reference is defined as the indices of nonlocal data accessed by processor i and owned by processor j. Similarly, OUT_{ij} of a reference is defined as indices of the data owned by processor i and needed by processor j. These definitions imply $\mathbf{IN}_{ij}$ and $\mathbf{OUT}_{ji}$ are equal. Let m_{kl} be a message to be rerouted from processor k to processor l and i_1, i_2,...,i_m be the intermediate processors. The OUT_{kl} due to the reference is replaced by the following sets OUT_{ki}, $\mathrm{T\_OUT}_{i_1 i_2}$, $\mathrm{T\_OUT}_{i_2 i_3}$,..., $\mathrm{T\_OUT}_{i_m l}$, where T_OUTs are the OUT sets for the rerouted data. Similarly, IN_{lk} is replaced by IN_{li_m},..., $\mathrm{T\_IN}_{i_2 i_1}$,$\mathrm{T\_IN}_{i_1 k}$. During the code optimization phase temporary buffers for the rerouted messages are allocated and the messages are combined with others during the message aggregation optimization.

```
for i = ...                          for i = ...
        ...                                  ...
        A(i) = ···+ A(i-d_i)+···     E_1(i) = A(i-d_1)
        ...                          E_2(i) = E_1(i-d_2)
endfor                                       .
                                             .
                                     E_{n-1}(i) = E_{n-2}(i-d_{n-1})
                                     A(i) = ···+E_{n-1}(i-d_n)+···
                                             ...
                                     endfor
```

Figure 6.10: Source code transformation for A(i-d_i)

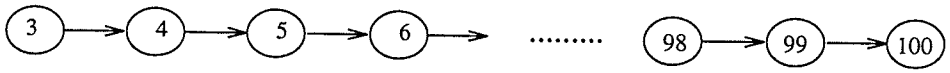
6.4 Program Transformation

To incorporate this scheme the algorithms for communication analysis should be modified. An alternative is to introduce a source to source transformation after the program analysis phase. This approach alleviates the communication analysis phase from the additional work required for rerouting. This could be accomplished by a program transformation that uses distributed arrays for buffering rerouted messages and additional assignments for rerouting messages. Let $d_i = d_1 + d_2 + \cdots + d_n$ be a transitive dependence and the corresponding reference be A(i-d_i). The program transformation allocates buffer space by declaring E_1, E_2, $\cdots$, E_{n-1} as arrays with size and distribution identical to that of A. The reference A(i-d_i) is replaced by a sequence of assignment statements shown in Figure 6.10.

Recall the ISDG of Figure 6.2(c). The dependence distance 2 is identified as a transitive-edge dependence so it can be replaced by an alternative path (1,1). Figure 6.11(a) illustrates the source code modification of the program shown in Fig-

```
REAL X(100), E1(100)
PARAMETER (n$proc = 4)
DECOMPOSITION D(100)
ALIGN X, E1 WITH D
DISTRIBUTE D(CYCLIC)
E1(2) = X(1)
do k= 3, 100
        E1(k) = X(k-1)
        X(k) = X(k-1)+E1(k-1)
enddo
```

(a) Fortran D Program After Code Transformation for the Program in Figure 6.2(a)



(b) ISDG of the Loop in (a)

Figure 6.11: Example of Source Code Transformation for Nonnested Loop

ure 6.2(a). Array E_1 is used to store the message to be rerouted. The size and distribution of E_1 are identical to that of array A. The ISDG of the modified program is shown in Figure 6.11(b). The edges for transitive-edge dependences have been removed from this ISDG. Figure 6.12 gives the SPMD code generated for the modified code.

This thesis focuses on the array dimensions with **cyclic** data distribution. For block-distributed arrays, the transitive-edge dependences may not cause transitive interprocessor communication because the source and sink of the dependence are executed in the same processor or across two adjacent processors. Consider the example in Figure 6.13. Assume that A is block-distributed among 4 processors. Dependence distance 4 is a transitive-edge dependence and its alternate path is (2,2). Because of the block distribution, the messages for loop-carried dependences 2 permits fewer messages to be aggregated at commlevel i so they are sent

```
{* compiling result *}
REAL X(25), E1(25), temp
{* next : the succeeding processor whose PID is ((my$proc+1) mod n$proc) *}
{* prev : the preceding processor whose PID is ((my$proc-1+n$proc) mod n$proc) *}
my$proc = myproc()
lb1=1
ub1=25
if (my$proc < 2) lb1 = 2 {* boundary condition *}
if my$proc=0 send(X(1),next)
if my$proc=1
        recv(E1(1),prev)
        send(X(1),E1(1),next)
endif
do k= lb1, ub1
        recv(E1(k), temp, prev)
        X(k)=E1(k)+temp
        send(X(k),E1(k), next)
enddo
```

Figure 6.12: Compiler Output

```
REAL A(100),...
DECOMPOSITION D(100)
ALIGN A with D
for i = ...
    ...
    A(i) = ···+ A(i-2)+A(i-4)+···
    ...
endfor
```

Figure 6.13: Example of Fortran D Program Segment

to their immediate successors. For block-distributed arrays, the transitive-edge dependences may not cause transitive interprocessor communication because the source and sink iterations of the dependence are executed in the same processor or across two adjacent processors. To apply the optimization effectively for a regular nonnested loop, the number of processors must be greater than the maximum of the dependence distances. If the number of processors is less than the maximum of the dependence distances, then the transitive-edge dependences may not cause communication. In Figure 6.13, if A is distributed cyclicly among two processors, then the source and the sink of the transitive-edge dependence are executed in the same processor. In this case, interprocessor communication is not required.

This phase is identical for regular double-nested loops and non-nested loops, except each transitive-edge dependence d_i is a dependence vector with two components (d_{i1},d_{i2}) and the array A which causes the transitive-edge dependence is a 2-dimensional array. Let $d_i = d_1 + d_2 + \cdots + d_n$ be a transitive dependence and the corresponding reference be A(i-d_{i1},j-d_{i2}). The program transformation allocates buffer space by declaring E_1, E_2, $\cdots$, E_{n-1} as arrays with size and distribution iden-

```
for i = ...                                   for i = ...
        for j = ...                                   for j = ...
            ...                                           ...
            A(i,j) = ···+ A(i-d_i1,j-d_i2)+···            E_1(i,j) = A(i-d_11,j-d_12)
            ...                                           E_2(i,j) = E_1(i-d_21,j-d_22)
        endfor                                                .
endfor                                                        .
                                                              .
                                                      E_n-1(i,j) = E_n-2(i-d_(n-1)1,j-d_(n-1)2)
                                                      A(i,j) = ···+E_n-1(i-d_n1,j-d_n2)+···
                                                      ...
                                                  endfor
                                              endfor
```

Figure 6.14: Source Code Transformation for A(i-di1,j-di2)

tical to that of A. The reference A(i-d_{i1},j-d_{i2}) is replaced by a sequence of assignment statements shown in Figure 6.14.

Recall that in Figure 6.7(c) the dependence distance vector (2,3) is identified as a transitive edge dependence and its alternative path is ((1,1),(1,2)). Figure 6.15 is the source-code modification of the program in Figure 6.7(a). In Figure 6.15 we define one more array E_1 with the same size and data distribution as array A and change the source code as described above. After this transformation, the transitive edges in the original ISDG disappear and the number of edges in the new ISDG is reduced. Figure 6.16 and 6.17 are the source modifications for Figure 6.8(a) and Figure 6.9(a), respectively. They all applied the same steps illustrated in Figure 6.15. Each example demonstrates the different cases of the second component of dependence distance vectors.

Any array in the above examples is cyclically distributed at least across one di-

```
A(100,100),B(100,100), E1(100,100)
Parameter (n$proc = 100)
Decomposition D(100,100)
Align A,B,E1 with D
Distribute D(CYCLIC,BLOCK)
E1(2,2) = A(1,1)
do i = 3 , 100
    do j = 4 , 100
        E1(i,j)= A(i-1,j-1)
        A(i,j) = B(i-1,j-1)
        B(i,j) = B(i-1,j-2) + E1(i-1,j-2)
    end
end
```

Figure 6.15: Code Transformation for the Program in Case One

mension. If an array is (BLOCK,BLOCK) distributed, this optimization would not be applicable. Similar to one dimensional array with BLOCK distribution, if an array is (BLOCK,BLOCK) distributed, the source and sink iterations of a transitive dependence are executed by the same processor, except for the iterations that access the boundary elements of the data space aligned to a processor. When a two dimensional array is distributed across processors they are configured in two dimensions. In other words, if the total number of processors, $m*n$, is configured as an $m*n$ grid. To apply this optimization to (CYCLIC,CYCLIC) distribution, the value of m and n should be greater than the largest absolute value of the corresponding components of the dependence distance vectors. This guarantees that iterations corresponding to the source and sink of any dependence are executed on two different processors. For cases 1 and 2, if the array is (BLOCK,CYCLIC) or (CYCLIC,BLOCK) distributed, the largest value of the dependence distance vector among the absolute value of the dependence distance vector components

```
A(100,100),B(100,100), E1(100,100)
Parameter (n$proc = 100)
Decomposition D(100,100)
Align A,B,E1 with D
Distribute D(BLOCK,CYCLIC)
E1(2,2)= A(1,4)
do i = 3 , 100
    do j = 1 , 100
        E1(i,j)= A(i-1,j+2)
        A(i,j) = B(i-1,j+2)
        B(i,j) = B(i-1,j+1) + E1(i-1,j+1)
    end
end
```

Figure 6.16: Code Transformation for the Program in Case Two

corresponding to the cyclical distributed dimension should be less than the dimensionality of the processor grid in that dimension. The same conditions applies to (*,CYCLIC) and (*,BLOCK) distributions too. (BLOCK,*) and (*,BLOCK) are special cases of (BLOCK,BLOCK) distribution. For case 3, this optimization can only be applied to (CYCLIC,CYCLIC) distribution.

In all the three examples, the array A and B were given a different data distribution. However, in each case, when given at least one dimension with cyclic distribution, this algorithm reduces the number of messages caused by transitive-edges.

```
A(100,100),B(100,100), E1(100,100)
Parameter (n$proc = 100)
Decomposition D(100,100)
Align A,B,E1 with D
Distribute D(CYCLIC,CYCLIC)
E1(2,5)= A(1,4)
do i = 3 , 100
    do j = 2 , 97
        E1(i,j)= A(i-1,j-1)
        A(i,j) = B(i-1,j-1)
        B(i,j) = B(i-1,j+3) + E1(i-1,j+3)
    end
end
```

Figure 6.17: Code Transformation for the Program in Case Three

6.5 Analytical Evaluation

This section presents an analytical evaluation of the proposed optimization . In general, the cost of a message transfer is dependent on the interconnection and the routing schemes employed in addition to the communication volume. Since startup costs outweigh other costs and communication volume is sparse, only a simple model for communication overhead is required. Recall the communication overhead is divided into three components: T_{start} - time required to setup the *send* and *recv* instructions, T_{copy} - time required to copy a message from program address space to the system buffer space, and $T_{transit}$ - time required to transmit a message through the physical medium.

To simplify the analysis the alternate path has exactly one intermediate node. Let P_1, P_2, and P_3 be the source, the intermediate, and the sink processors of

the message to be rerouted. Let $T_{compute}$ be the computation time in P_2 before it begins sending the combined message. Since a processor sends/receives messages sequentially and $T_{compute}$ is less than T_{start}, the time elapsed from the time P_1 begins to send the first message to the time P_3 received both the messages is $3(T_{start}+T_{copy}+T_{transit})$. With the optimization, only two messages are exchanged but the time required to copy or transmit a message is doubled because the message size is doubled. The total time elapsed from the first message sent by P_1 to the time P_3 receives the combined message is $2\,T_{start}+4\,T_{copy}+4\,T_{transit}+T_{compute}$. Hence, this optimization would reduce the communication overhead if $(T_{copy}+T_{transit}+T_{compute}) < T_{start}$.When the number of intermediate nodes increases, the left hand side of inequality would increase linearly and there is a threshold beyond which this optimization is not feasible.

6.6 Conclusions

We have presented an efficient technique for rerouting the messages for transitive-edge dependences with loops having constant dependence vectors. This method reduces the number of messages by analyzing communication pattern among all the processors in a pipelined computation, which is different from some traditional compiler optimizations that focus on identifying the communication between any two processors. This technique can be implemented as a phase before communication optimization of the FORTRAN D compiler. Then all messages for the transitive dependences will be combined to the messages on the alternative path after the compiler applies message aggregation optimization.

Chapter 7

Conclusions

Programming on parallel computers is always considered as a tedious job. Programmers are required to have some knowledge of the underlying machine architectures. Scientists are attempting to develop a machine-independent programming environment. FORTRAN D is one of these languages. In order to increase the reusability of the existing conventional software, the research group of FORTRAN D enhanced FORTRAN with a rich set of data decompositions. The FORTRAN D compiler is designed to generate data-parallel programs which can be efficiently executed on the nodes of a distributed-memory machine. To improve the run-time performance reducing communication overhead in the compiler output is a major goal of the FORTRAN D compiler.

This thesis presented a method for reducing communication overhead in the compiler output of a parallelizing compiler. This method is presented in the framework of FORTRAN D programming environment. It reduces the communication messages by analyzing the dependence structure of the source program. First, it char-

acterizes the dependence structure of the FORTRAN D program into a iteration space dependence graph (ISDG), then by analyzing the ISDG this method identifies the transitive-edge dependences which can be removed later from the dependence structure. Second, by performing a source-to-source transformation to the source program the corresponding messages for the transitive edges are efficiently rerouted in the compiler output. After the transformation, the transitive-edge dependences identified in the first step do not exist in the dependence structure of the transformed program.

This method reduces the number of messages by analyzing communication pattern among all the processors in a parallel computation which is different from some traditional compiler optimizations that only focus on identifying communication between any two processors. Since the dependence analysis is included in the FORTRAN D programming environment, this method can be incorporated as a phase after the dependence analysis and before communication optimization in the FORTRAN D compiler. Finally, all the messages for the routed transitive-edge dependences can be combined after the compiler applies message aggregation optimization.

The thesis focuses on identifying the transitive edges of the ISDGs of single loops and double-nested loops with constant dependence vectors. This optimization provides an efficient way to reduce the number of messages. The analytical evaluation of the proposed scheme demonstrated that this scheme would reduce the communication overhead in the compiler output. The algorithm for identifying transitive-edge dependences can also be applied to shared-memory multiprocessors. In a shared-memory programming environment, this algorithm can be used to remove

redundant synchronization instructions in the compiler output. Those redundant synchronization instructions are transitive-edge dependences in the ISDG(s). This algorithm can be incorporated in a parallelizing compiler for shared-memory machines to reduce the synchronizing computation in the compiler output.

7.1 Future Work

The following are some areas for continuing the research:

Experimental Evaluation and Validation on a Distributed-Memory Machine : to implement this optimization and incorporate it into a parallelizing compiler; this can be done on a distributed-memory machine, such as IBM SP2.

Extending the Optimization to higher Dimensions : to extend the the research of identifying the transitive edges of ISDGs of higher dimensions.

Reducing Storage Buffer for Rerouted Messages : to reduce the size of the temporary buffers which are used to store the rerouted messages for the transitive-edge dependences in the transformed program.

Bibliography

[1] Huang, K., *Advanced Computer Architecture: parallelism, scalability, programmability*, McGraw-Hill, Computer Science Series, 1993.

[2] Akl, S.G., *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Englewood Cliffs, N.J., 1989

[3] Hatcher, P.J., and Quinn, M.J. *Data-Parallel Programming*, The MIT press, 1991.

[4] Quinn, M.J. *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill Series in Supercomputing and Artificial Intelligence, 1987.

[5] Kuskin, J., Ofelt, D., Heinrich, M., and Heinlein, J. "The Stanford FLASH Multiprocessor", *In Proceedings of the 21st International Symposium on Computer Architecture*, Chicago, IL, April 1994, pp 302-313.

[6] Torrellas, J., Xia, C., and Daigle, R. " Optimizing Instruction Cache Performance for Operating System Intensive Workloads", *1st International Symposium on High Performance Computer Architecture*, 1995, pp 360-369.

[7] Stallard, P. W. A, Muller, L. and Warren, H. D., "Performance Evaluation of Parallel Programs on the Data Diffusion Machine". *In Performance Evaluation*

of Parallel Systems, PEPS '93. University of Warwick, UK, November 1993, pp 94-101.

[8] Fox, G., Hiranandani, S., and Kennedy, K., FORTRAN D Language Specification, Technical Report TR90-141, Department of Computer Science, Rice University, December 1990.

[9] Banerjee, U.,*Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, 1988.

[10] *IBM AIX Parallel Environment Parallel Programming Primer*, Release 2.0. IBM Corporation, 1994. (http://ibm.tc.cornell.edu/ibm/pps/doc).

[11] Wolfe, M., *Optimizing Supercompilers for Supercomputers*, The MIT Press, 1989.

[12] Krothapalli, V.P. and P. Sadayappan. "Removal of Redundant Dependence in DOACCROSS Loops with Constant Dependences", *IEEE Transactions on Parallel and Distributed Systems* July 1991, pp 281-289.

[13] Tseng, C. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*, Ph. D Thesis, Rice University, January 1993.

[14] Hiranandani, S., Kennedy,K. and Tseng, C. "Evaluation of Compiler Optimizations for Fortran D Compiler on MIMD Distributed-Memory Machines", In *Proceedings of the 1992 ACM international Conference on Supercomputing*, Washington, DC, July 1992, pp 1-30.

[15] Zima, H., H.-J. Best, and M. Gerndt. "SUPERB: A tool for semi-automatic MIMD/SIMD parallelization" *Parallel Computing*, Vol.6, No:1, January 1988, pp 1-18.

[16] Koelbel, C., Mehrota, P. "Compiling Global Name-Space Parallel Loops for Distributed Execution", *IEEE Transactions on Parallel and Distributed Systems*, Vol.2, No:4, October 1991, pp 440-451.

[17] Thinking Machine Corporation, Cambridge, MA. *CM FORTRAN Reference Manual*, Version 1.0., Cambridge, MA 02142, February 1991 .