The Role of Performance Models in Parallel Programming and Languages

by

Ton Anh Ngo

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1997

Approved by_____
(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree_____

Date_____

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 300 North Zeeb Road, Ann Arbor Michigan 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

The Role of Performance Models in Parallel Programming and Languages

by Ton Anh Ngo

Chairperson of the Supervisory Committee: Professor Lawrence Snyder
Department of Computer Science
and Engineering

A program must be portable, easy to write and yield high performance. Unfortunately, these three qualities present conflicting goals that require difficult and delicate balancing, and for parallel programs, the parallelism adds yet another level of complexity. The difficulty has proved to be a persistent obstacle to parallel systems, even for the data parallel class of applications where the parallelism is abundant and the computation is highly regular. This thesis is an effort to find this delicate balance between scalability, portability and convenience in parallel programming. To establish the framework for analyzing the many disparate and conflicting issues, I develop the concept of *modeling* and apply it throughout the study. In addition, the empirical nature of the problem calls for experimental comparisons across current parallel machines, languages and compilers.

The nonshared memory model is first shown to be more portable and scalable because it exhibits better locality, then two data parallel languages, HPF and ZPL, are studied in detail. HPF offers many advantages and enjoys wide attention in industry and academia, but HPF suffers from a significant gap in its performance model that forces

the user to rely completely on the optimization capability of the compiler, which in turn is nonportable.

On the other hand, ZPL's foundation in a programming model leads to predictable language behavior, allowing the user to reliably choose the best algorithm and implementation. In this respect, a new language abstraction is proposed that promotes scalability and convenient programming: *mighty scan* generalizes the parallel prefix operation to apply to conceptually sequential computation that is difficult to parallelize.

Table of Contents

List of Figures v

List of Tables vii

Chapter 1: Introduction 1

1.1 Modeling 1

1.2 Parallel programming and modeling 3

1.2.1 Model for machines 7

1.2.2 Model for compilers 10

1.2.3 Model for languages 12

1.2.4 The larger picture 13

1.3 Thesis outline 13

Chapter 2: Programming Models for Data Parallel Problems 17

2.1 Introduction 17

2.2 Methodology 20

2.2.1 The Machines 20

2.2.2 The applications and the implementations 23

2.3 LU Decomposition 25

2.3.1 The problem 26

2.3.2 The parallel algorithms 27

2.3.3 Volume of data references 29

2.3.4 Performance results 32
2.4 Molecular Dynamics Simulation 34
2.4.1 The problem 34
2.4.2 The algorithm 37
2.4.3 Volume of data references 41
2.4.4 Performance results 44
2.5 Conclusion 45

Chapter 3: Two Data Parallel Languages, HPF and ZPL 53
3.1 Introduction 53
3.2 A review of HPF 56
3.3 A review of ZPL 58
3.4 Expressing parallelism 61
3.4.1 Parallel computation 61
3.4.2 Array reference 64
3.5 Distributing parallelism 66
3.5.1 Distributing the data 66
3.5.2 Distributing the computation 73
3.6 Conclusion 75

Chapter 4: The Performance Model in HPF and ZPL 78
4.1 Introduction 78
4.1.1 The performance model 79
4.1.2 ZPL's performance model 81
4.1.3 HPF's performance model 82
4.1.4 A methodology to evaluate the performance model 83
4.2 Selecting an implementation: array assignment 86
4.2.1 Quantifying the performance model 86
4.2.2 Results 88

4.3 Selecting an algorithm: matrix multiplication 98
4.3.1 HPF versions 98
4.3.2 ZPL versions 100
4.3.3 Results 101
4.4 Current HPF solutions 108
4.5 Conclusions 109

Chapter 5: Benchmark Comparison 111
5.1 Introduction 111
5.2 Methodology 113
5.2.1 Overall approach 113
5.2.2 Benchmark selection 115
5.2.3 Platform 116
5.2.4 Embarrassingly Parallel 118
5.2.5 Multigrid 120
5.2.6 Fourier Transform 124
5.3 Parallel Performance 126
5.3.1 NAS EP benchmark 126
5.3.2 NAS MG benchmark 131
5.3.3 NAS FT benchmark 133
5.3.4 Communication 134
5.3.5 Data Dependences 135
5.4 Conclusion 137

Chapter 6: Mighty Scan, parallelizing sequential computation 140
6.1 Introduction 140
6.2 A case study 142
6.2.1 Idealized execution 143
6.2.2 An algorithmic approach 144

6.2.3 Pipelining 147
6.3 Implementations by HPF and ZPL 149
6.3.1 DO loop implementation 149
6.3.2 An array oriented approach: F90 and ZPL 152
6.4 A new construct for ZPL 153
6.5 Conclusions 155

Chapter 7: Conclusions 157
7.1 Contributions 157
7.2 Summary 158
7.3 Future works 161

Bibliography 162

Appendix A: Performance data 174
A.1 LU on shared-memory machines 174
A.2 WATER on shared-memory machines 176
A.3 HPF and ZPL programs on nonshared-memory machine 177

List of Figures

1.1 A user perspective of parallel system 5

1.2 Realizing a parallel programming model 8

1.3 Experimental comparisons of the programming models and languages. . . 15

2.1 Machine memory hierarchy 20

2.2 LU speedup based on model: n=512, t=4 32

2.3 LU execution time: Sequent and Cedar 35

2.4 LU execution time: Butterfly and KSR 36

2.5 LU execution time: DASH 37

2.6 LU speedup: Sequent and Cedar 38

2.7 LU speedup: Butterfly and KSR 39

2.8 LU speedup: DASH 40

2.9 WATER speedup based on model 43

2.10 WATER execution time: Sequent and Cedar 46

2.11 WATER execution time: Butterfly and KSR 47

2.12 WATER execution time: DASH 48

2.13 WATER speedup: Sequent and Cedar 49

2.14 WATER speedup: Butterfly and KSR 50

2.15 WATER speedup: DASH 51

2.16 $\frac{P_{ns}}{P_s}$ factor for all machines 52

3.1 Reference to a distributed array in ZPL. 65

3.2 Example of HPF data layout. 68
3.3 Example of ZPL data layout. 70

4.1 Cross compiler performance for HPF array assignment. 90
4.2 Communication for array assignment using different index expressions: p=8. 93
4.3 Execution time for array assignment using different index expressions: p=8 95
4.4 Matrix Multiplications by ZPL and 3 HPF compilers: 2000x2000, p=16 . 102

5.1 Illustrations of EP as implemented in HPF and ZPL. 121
5.2 Illustrations for the Multigrid algorithm. 123
5.3 Illustrations of FT implementation. 125
5.4 Performance for EP (log scale). 127
5.5 Performance for MG (log scale). 128
5.6 Performance for FT(log scale). 129

6.1 Two methods for array partitioning. 142
6.2 Three possible parallel executions. 144

List of Tables

2.1 Characteristics of the shared memory machines 21

2.2 Volume of data references for LU 31

2.3 Volume of data references for WATER 42

3.1 High level comparison of HPF and ZPL 55

4.1 Array assignment in HPF and ZPL 89

4.2 Choices of array assignment in HPF. 92

4.3 Matrix multiplication algorithms expressed in HPF. 99

4.4 Matrix multiplication algorithms expressed in ZPL. 101

4.5 Pseudo-code for the matrix multiplication algorithms by three HPF compilers. 103

4.6 Speedup ratio from 16 to 64 processors, 2000×2000 matrix multiplication.107

5.1 Sources of NAS benchmarks for HPF and ZPL 117

5.2 Communication statistics for EP class A, MG class S and FT class S: p=8 135

5.3 Dependence ratio $\frac{m}{n}$ for EP, MG and FT. 137

6.1 The forward elimination step from tomcatv. 150

6.2 Pseudo-code for the tomcatv segment by HPF and ZPL compilers. 151

6.3 tomcatv expressed using SCAN and the resulting SPMD code. 154

A.1 Performance (seconds) of LU Decomposition on 5 shared memory machines.174

A.2 Performance (seconds) of WATER on 5 shared memory machines. 176
A.3 Characteristics of the IBM SP2 used for the cross-compiler comparison . . 178
A.4 Performance (seconds) of Matrix Multiplication on the IBM SP2: 2000×2000 178
A.5 Performance (seconds) of NAS 2.1 benchmarks on the IBM SP2: EP, FT, MG. 179

Acknowledgments

This dissertation would not have been possible without the ever ready encouragement and guidance from my adviser, Lawrence Snyder.

I am also indebted to the IBM Thomas Watson Research Laboratory, Yorktown Heights, NY for supporting my entire graduate program, and to my managers Fran Allen, Kevin McAuliffe, William Brantley, Edith Schonberg, Lee Nackman, Jim Russell and John Barton.

The experimental work in this thesis required the parallel computing resources of the Lawrence Livermore National Laboratory, Stanford University, the University of Illinois at Urbana-Champaign, the Cornell Theory Center, the Caltech Center for Advanced Computing Research and the San Diego Supercomputing Center. I would like to express my gratitude to these institutes for making their resources available.

I also would like to thank my colleagues Manish Gupta and Bradford Chamberlain who have always been ready to help with issues on HPF and ZPL.

Dedications

My parents, Dinh Pham and Bao Ngo, and my brothers and sisters Tuan, Dung, Tu and Thu have always been supportive in quietly reminding me that I do need to finish my graduate study some time. Dung has been especially stimulating despite the extremely difficult life challenge he is facing. To them, I dedicate this work.

My wife, Hue, and I have found that medical school, graduate school and children all at once is not exactly fun, but along our arduous journey together, we have discovered a strong sense of closeness. Nevertheless, we would not recommend to anyone to try it, at least not without a lot of help and a healthy sense of humor.

Chapter 1

Introduction

model 1(n) (Webster 7th dictionary)
DEFINITIONS:

[1 (obs)] a set of plans for a building
[2 (dial Brit)] COPY, IMAGE
[3] structural design
[4] a miniature representation of something;
also : a pattern of something to be made
[5] an example for imitation or emulation
[6] a person or thing that serves as a pattern for an
artist; esp : one who poses for an artist
[7] ARCHETYPE
[8] one who is employed to display clothes or other merchandise:
MANNEQUIN
[9a] a type or design of clothing
[9b] a type or design of product (as a car or airplane)
[10] a description or analogy used to help visualize something
(as an atom) that cannot be directly observed
[11] a system of postulates, data, and inferences presented as a
mathematical description of an entity or state of affairs

1.1 Modeling

This thesis is based on the concept of modeling. Although the term *model* is heavily used in many fields and our focus in this study will be quite narrow in contrast to this general term, it is helpful to remember that it forms the underlying basis for this work.

In the simplest terms, a model reflects our understanding of how a certain system operates; it captures the information that is necessary and sufficient for our expected use of the system. For instance, when we operate an automobile, we have in mind a certain model of the machinery: we expect features such as a steering wheel, an accelerator pedal, a brake pedal, etc. We expect these features to function in a certain manner: the accelerator should move the car and the brake should stop it. We operate these features and indeed they function as expected. We are also conscious of the size of the car, i.e., how far ahead and behind are the front and back bumpers, so that we can avoid colliding with other cars. This conformity to a consistent model allows a person to step into a car of any make or model and drive with no difficulty.

A model may capture different levels of detail depending on the needs of the user. For instance, an auto mechanic may look at an automobile and think in terms of the engine, transmission, carburetor, electrical wiring, while a car salesman may think in terms of front-wheel drive, antilock brakes, and service contract.

Furthermore, the model must be accurate for the system to be operated effectively. Consider a race car driver whose goal is to drive as fast as possible without wrecking the car. Among other things, he must learn precisely how the car steering responds at high speeds, or more formally, he must acquire in his mind an accurate model of the car's steering behavior. A number of scenarios can occur that prevent such a mental model: (1) the steering is soft because of the car's poor design, (2) some last minute repair work had altered the steering and the instruction given to the driver concerning the change turns out to be inaccurate, or (3) the car is new and he has not gained enough experience with its behavior. Clearly, if the driver is not sure how the car will steer at high speeds, he will have to be conservative and drive at a lower speed.

Modeling is also pervasive in computer science. An example of a successful model can be found in the design of the memory for a sequential machine. Computer architects effectively model a program as a stream of references that exhibit some temporal and spatial locality. Designers then optimize to this model, and the result is the cache, a

ubiquitous and very successful feature.

Interestingly, this example can be carried further to illustrate an instance when the model fails. A cache under the expected operating conditions will yield an average memory access time that is much better than that of the main memory, thereby creating an illusion (i.e., model) of a uniformly fast memory. A user then can program according to this model by largely ignoring the memory access time. However, unfavorable behaviors can occur, such as a long sequence of consecutive addresses or a stride in the references which makes use of only one word per cache line. In these cases, the original model of temporal and spatial locality fails, resulting in the failure of the cache and consequently of the user perception of a uniformly fast memory. The user then encounters surprisingly poor performance. Fortunately in this particular case, cache blocking by the compiler has been successful in identifying these reference patterns and breaking the long sequence or adjusting the memory stride. In other words, the compiler is able to transform the program so that it fits the original model on which the cache design is based.

In this thesis, I will focus on the topics of programming for parallel machines, and the modeling concept is used to study the issues and problems.

1.2 Parallel programming and modeling

Parallel systems are among the few disciplines in computer science in which very substantial benefits appear feasible but have remained elusive despite extensive research. Consequently, it is not surprising that the interest in parallel systems has waxed and waned over the years. The late 80's and the early 90's saw a surge in the development of parallel machines and software, driven by the availability of cheap microprocessors. In the last few years however the number of vendors as well as the variety of parallel machines has been dwindling. The level of interest in parallel systems at the current time seems low, yet the relentless quest for more computing power has not eased and it is conceptually clear that parallel machines can lead to faster execution times than a sequential machine. This motivates the question, "What are the major obstacles in

realizing the potential of parallel systems?" In seeking an answer, it is helpful to consider a more elementary question, "What does a user expect from a parallel machine?" The following expectations are identified.

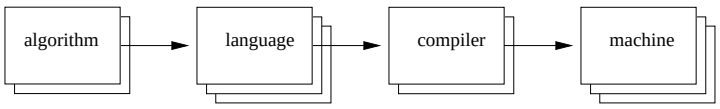
Foremost is performance, or more precisely *scalable performance*, the principal benefit from a parallel machine. It is clear that a user who considers a parallel machine does so out of necessity. Without scalable performance, there is little motivation for the user to depart from the familiar sequential machine. Scalable performance has two components: (1) competitive scalar performance against a state of the art sequential machine, and (2) good speedup.

The second expectation from users is the durability of their software. Program development is costly without the added difficulty of parallel programming; therefore, the users expect their programs to be *portable* not only across different parallel platforms but also across generations of parallel machines. It should be stressed that portability must include both correctness *and* performance; otherwise, a correct but slow program is no more useful than a fast program that produces incorrect results.

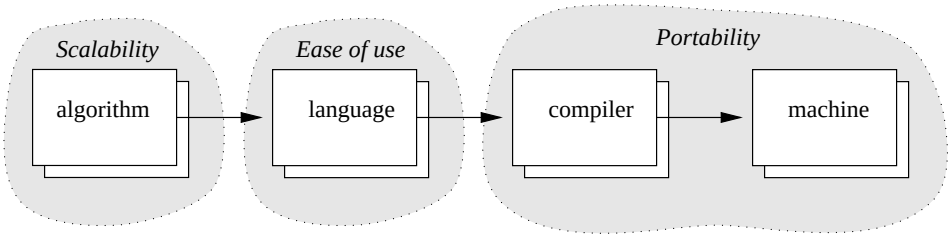
The third expectation is *ease of use*. This may seem to have lower priority than the first two, but it is clear that parallel machines will not gain a critical mass of users until they can be made readily accessible. We next consider how various aspects of parallel systems are meeting these requirements.

A contract between the user and the system

Figure 1.1(a) shows a user's perspective in developing a parallel program. From the problem specification, a user typically chooses an algorithm, implements it in a particular language, compiles the program with a particular compiler, then runs the program on a parallel machine. On a closer look (Figure 1.1(b)), we can see that the portability issue mainly involves the compilers and the parallel machines since the goal in portability is to be able to run the same program without modifications on different platforms. The language is only involved in the sense that one standard language is used. The ease of use issue falls entirely within the language since it depends on the type of abstractions



(a) Developing a parallel program



(b) Satisfying the three user expectations

Figure 1.1: A user perspective of parallel system

for parallelism that the language provides.

Scalability ultimately rests with the user in the choice of the algorithm. Since the language is the only interface accessible to the user, he/she would learn the abstractions and functionality provided by the language and, in the process, form a mental picture of how the parallel execution is to proceed. This mental picture, or more formally the *programming model*, is then used to select the best algorithm for the problem and to implement the algorithm in the most efficient manner. The responsibility of the compiler and the machine is to accurately implement the programming model that that language presents. The programming model thus serves as the *contract* between the user and the parallel system. It follows that if either of the parties of the contract fails, e.g. the user fails to find a scalable solution or the system fails to meet the expectation, then scalability is not achieved.

As an example, consider the case of KAP, a parallelizing compiler for Fortran programs. A KAP user would program in the standard Fortran syntax with the understand-

ing that when a loop is encountered, a number of processors will wake up and execute the loop in parallel. Under this programming model, the user will choose and implement an algorithm based on a number of assumptions: (1) the loop that is expected to execute in parallel will indeed execute in parallel, (2) the overhead for entering the parallel execution is negligible, and (3) the data can be accessed with an average low latency. Then, the task for the user is to ensure that the DO loop dominates the program execution, while the task for the parallelizing compiler is to deliver the expected performance. If the compiler meets the expectation, then the program scales; otherwise, the program does not scale. In the latter case, the failure is not due to the algorithm since it is indeed scalable according the programming model; rather, the failure rests with the compiler in implementing the model.

The glue for the components of a system

Thus far we have only discussed modeling as an interface between the user and the language, yet modeling exists between other components as well. Consider the models in a native system as illustrated in Figure 1.2(a). In the development of many parallel systems, the effort has tended to focus on the hardware, leaving the software often under-developed and primitive. In addition, machine vendors are often motivated toward proprietary software that closely mirrors the functionality of the hardware. The result is a tight coupling between the machine, the compiler, and the associated language. In other words, the machine is designed for a particular architecture, the language is designed to match the machine capability, and the compiler translates directly from the language syntax to the machine function. The models that exist between these components contain specific details that help make the system efficient. However a disadvantage is that programs tuned to the system may not be portable. An example is the Connection Machine CM-2 and its C* language: the *where* construct is an excellent match for the SIMD control mechanism in the hardware, but implementing this language abstraction on an MIMD may involve more overhead.

Figure 1.2(b) shows a more loosely coupled scheme in which a language, preferrably

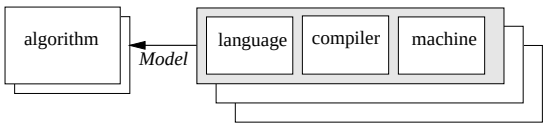
a standard one, is targeted by many compilers. The compiler in turn can target a number of parallel machines. In this approach, a language designer must make a number of assumptions about the capability of the compilers, i.e. a compiler model. Likewise, a compiler developer who intends to port the compiler implementation to multiple machines must have a certain model common to all the machines. A compiler for distributed memory machines could expect the machine to provide common communication functions such as asynchronous send/receive or even a standard communication interface such as MPI.

The decoupling of the components renders the modeling aspect more critical because a mismatch in the chain of models has the potential of degrading the scalability of the entire system. Accurate modeling thus becomes the "glue" that ties together the components of a fully portable parallel system. Clearly, any reasonable model can be used between any two components. The only requirement is that the model can be implemented faithfully, or conversely, the model accurately captures the performance of the components being modeled.

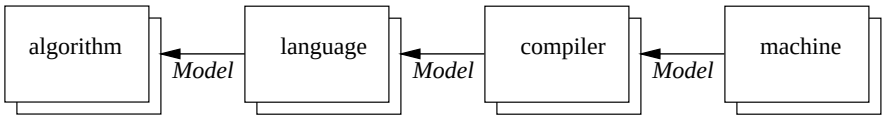
Having established the high level picture, we can summarize the overall approach for scalable, portable and easy to use systems: (1) the parallel language must be designed carefully for ease of use; (2) for portability, the language must be supported on multiple compilers and machines; (3) for scalability, the modeling must be kept accurate across all components of the parallel system. We now briefly survey the state of the art in parallel machines, compilers and languages to understand the types of model that are in use.

1.2.1 Model for machines

The model at the machine level can typically be derived from a description of the machine architecture and organization. Parallel architectures proposed over the years cover a wide spectrum ranging from shared memory and distributed memory to data flow and multithreading. Since the processing elements are not different from the uniprocessors, the models for parallel machines are generally distinguished by the memory system:



(a) Native models: tight coupling between the machine, the compiler and the language leads to poor portability



(b) More portable models: each phase is decoupled

Figure 1.2: Realizing a parallel programming model

shared memory and nonshared memory. Note that we use the term *nonshared* instead of the more conventional term *distributed* to avoid confusion since a shared memory machine may have a distributed memory organization.

Shared memory architecture is characterized by a single address space for all processors, while in the nonshared memory architecture each processor explicitly manages its address space and shares data only through explicit messages that involve both the sender and the receiver. The shared memory model can be further qualified by the knowledge of the physical location of an address. In other words, a processor is able to directly access any memory address, but it may or may not be able to determine whether the address is local or nonlocal. This knowledge imparts on the shared memory model some characteristics of the nonshared memory model; therefore for our purpose, this specialized model is called a hybrid model.

Shared memory machines

An early memory model for shared memory parallel machines is the Parallel Random Access Memory model (PRAM), which extends the sequential Random Access Memory

model (RAM) by assuming that each processor can access any memory location in unit time. The simplification allows a programmer to reason about the complexity of different algorithms so that an efficient algorithm can be chosen. For the shared memory architectures, a number of machines present a pure shared memory model by implementing a memory with a uniform memory access time (UMA). These include symmetric multiprocessors based on a bus or crossbar (SMP, currently marketed by numerous vendors such as Sequent, SGI), as well as earlier machines that use a multistage interconnect network (MIN) to connect the processors to a global memory. In the actual implementation, the memory access time is not strictly uniform, but, as with the cache in a sequential machine, there is no direct method for the user to control the access time. Other architectures fall into the category of pure shared memory as well. Cache Only Memory Architecture (COMA) implemented in the Kendall Square Research machine does not allow the general user to differentiate among memory locations since the machine's responsibility is to dynamically relocate the memory sections in use to the local processor. Data flow machines such as the Monsoon machine[Papadopoulos & Culler 90] closely match the data flow programming model, which has no explicit notion of data location. Finally, the Tera machine relies on the program parallelism to mask the high latency of the global shared memory, so that if sufficient parallelism exists, the global memory will appear to be uniformly fast.

Hybrid machines

Large scale shared memory machines by necessity must distribute its physical memory. To take advantage of the processor locality, many machines provide some mechanisms for the user to distinguish different levels in the memory hierarchy, resulting in a model of nonuniform memory access time (NUMA). The IBM RP3 and BBN Butterfly distinguished local and remote addresses while the Stanford DASH could recognize node, cluster and remote memory addresses. Recent advances in network technology in reducing the latency and improving the bandwidth are also shifting the focus to clusters of SMP's. These clusters (Distributed Shared Memory, or DSM) are generally connected

through a high performance switch and have a hardware coherent cache. Thus departing from pure shared memory, these machines present a hybrid model that has both shared and nonshared memory characteristics: hardware for shared memory is provided, but the underlying communication mechanism is also exposed to varying degrees, which could be as simple as fetching a fixed size cache line, or more full-fledged with *put/get* of variable size messages (Cray T3D, T3E).

Nonshared memory machines

At the other end of the spectrum are the message passing machines with the pure nonshared memory model. Early machines such as the Caltech Hypercube support little more than the basic send and receive, but more recent machines (SP2, Paragon, CM-5) enrich the communication functionality to include asynchronous and collective communication. The level of communication tends to be coarse in most machines, thereby encouraging coarse grained parallelism, but fine grained communication at the level of program variables is also possible in machines such as the Transputer.

1.2.2 Model for compilers

A language designer must be aware of the capability of the compiler technology to ensure a viable implementation of the language. Within the framework in Figure 1.2(b), this awareness constitutes the compiler model. In general however, the model that compilers present is not commonly recognized as a separate entity. It is not always clear whether a language design is based on an existing compiler model or assumes a future compiler model with the expectation that new compiler optimizations will be developed. In the latter case, the language designer has in effect created and used a new model that some future compiler must satisfy. We briefly survey some current approaches in parallel compilers to understand the models in use.

When a parallel language is extended from a sequential language through libraries, no additional effort is required from the compiler. The compiler treats a parallel program no differently than a sequential program; therefore no compiler model exists with respect

to parallelism. Examples include C/Fortran compilers for shared memory programs that use the conventional thread abstractions (lock, spawn, etc.) as well as message passing programs that use the standard communication libraries (PVM, MPI, etc.).

Compilers for sequential languages typically optimize for locality in the spatial and temporal dimensions of the reference pattern. Compilers for shared memory parallel languages must deal with the additional processor dimension in the reference pattern. On a cache-coherent shared memory machine, a compiler may manipulate the data sharing pattern, for instance by reorganizing the memory layout to minimize the false sharing effect[Jeremiassen & Eggers 95]. In this case, the compiler model would include the capability to identify the shared data, analyze the access patterns and perform the necessary transformations.

For parallelizing compilers, the model requires extensive capabilities. The compiler must be able to analyze and disambiguate dependences in the DO loops and perform the necessary transformations so that the loops can be executed in parallel. If the memory hierarchy is not uniform, the compiler must partition the data to maximize locality. The compiler must also minimize any overhead involved in creating the parallelism such as data copying, synchronization, managing worker threads, etc.

HPF and directive-based parallel Fortran variants assume a compiler able to exploit the data distribution directives. Since these languages mainly target distributed memory machines, the compiler must (1) analyze the index expression to determine whether each memory reference is local or remote, (2) generate the communication required, and most importantly, (3) optimize the communication so that the program performance is scalable. In addition, the standard Fortran semantics must be observed.

For ZPL[Lin & Snyder 93], the design of the language follows the philosophy that an efficient compiler implementation must exist for the abstractions. Although it seems contradictory to be proposing new powerful language abstractions that only require existing compiler optimization techniques, being designed from first principles and being developed incrementally allows the language to evolve with new compiler technology. In

this case, the compiler model is simply one that efficiently implements the language.

1.2.3 Model for languages

Models at the language level are also called programming models. They are interesting for several reasons. First, they are the direct interface between a user and the machine; therefore they must focus on the user's needs, e.g., ease of use. Second, it is easy to create and implement a new programming model because it can be easily interpreted in software. This has lead to a rich diversity of models, which allow for many ways to categorize them; however to be consistent with the machine model, we will classify them as shared-memory and nonshared-memory and discuss the variations within each class.

In the shared memory class, the most elementary model is one that directly reflects the functionality of the hardware. Control for hardware features such as lock, barrier synchronization, cache prefetch, etc. are coded in libraries which are then used to extend a sequential language. The model is the most efficient since it accurately captures the capabilities of the machine, but tends to be rigid. Differences in syntax and functionalities between machines also hamper program portability, although standard macros such as those from Argonne National Laboratory have been developed to ease this problem.

For languages targeted by parallelizing compilers such as KAP or PTRAN, the original sequential programming model is only extended with the awareness that parallelism will arise from the loops. A user would only need to ensure that loops constitute a major part of the dynamic execution of the otherwise sequential program.

Elementary nonshared memory languages are also extended from sequential languages with communication libraries. These libraries control the low level hardware and may add functionality such as message buffering, error recovery, and collective communication; therefore the programming model is usually richer than the actual machine. Differences in syntax and functionality lead to some portability problems, although they are alleviated to some degree by standard communication interface such as PVM and MPI.

Data parallel languages present an interesting model that has some characteristics of nonshared memory but allows on a global view of the program computation. Examples include C*, High Performance Fortran and similar variations, and ZPL. The nonshared memory quality can exist as data partitioning directives as with HPF, or can be implicit in the language abstractions as in ZPL. Under this model, the user programs with the knowledge that the data will be distributed, but is spared the tedium of coding low level communication to manage the distributed data.

1.2.4 The larger picture

Our discussion thus far has identified the following points:

1. Parallel programs must be scalable, portable and easy to write
2. Scalability originates in the choice of the algorithm
3. Ease of use is provided by the language
4. Portability requires the support of multiple compilers and machines
5. The appropriate models bind together the algorithm, the language, the compiler and the machine to form an effective system.

In this framework, a system as a whole is only as effective as the least effective link. This observation thus lays the basis for the methodology throughout the thesis: to evaluate a system, it suffices to evaluate the models that bind together the components. Evaluating the model in turn will involve formulating some questions that can be answered by experimental data.

1.3 Thesis outline

Having laid out the framework, the point of departure for this thesis can be stated as follows. Our goal is to understand the best approach for achieving scalability, portability

and ease of use in parallel programming. The scope is limited to the class of data parallel problems. The studies involve experimental comparison across parallel machines, compilers and languages. The concept of modeling serves as the basis for analyzing the observed behavior.

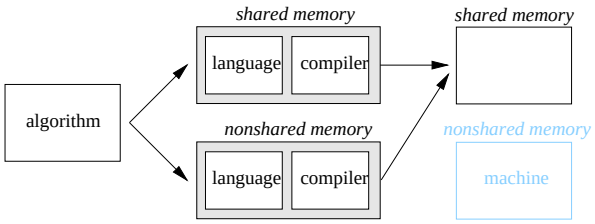
This thesis asserts that an accurate performance model is a fundamental requirement for a parallel system. It makes the following contributions:

- An experimental comparison and analysis of two general programming models: the shared and nonshared memory models.
- An experimental comparison and analysis of two data parallel languages: HPF and ZPL.
- A new high level data parallel abstraction that promotes scalability, portability and ease of use: Mighty scan.

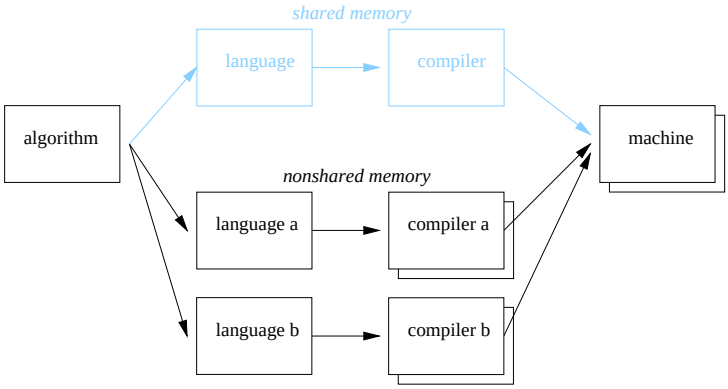
Chapter 2 begins with the issues of scalability and portability, omitting for the time the question of ease of use. Given the shared and nonshared memory programming models on different machines (Figure 1.3(a)), the question is which model is more portable across the machines. The data provides experimental evidence to support the choice of a nonshared memory programming model.

Having established a candidate programming model, we next consider the issue of ease of use. In the last few years, considerable research has focused on *high level* parallel languages that are based on the nonshared memory programming model and that target both *scalability* and *portability*. Although the data parallel model is not adequate to address general parallel programming, the prevalence of data parallel problems in scientific applications has motivated the development of many data parallel languages, all with the goal of enabling easier programming while achieving scalable and portable performance.

In this thesis we will consider HPF and ZPL (Figure 1.3(b)). Compilers for both HPF and ZPL are now widely available on many platforms. HPF vendors include APR (Applied Parallel Research), PGI (Portland Group Inc.), IBM, and DEC, and the supported



(a) Evaluating the portability of programming models



(b) Evaluating the portability of languages

Figure 1.3: Experimental comparisons of the programming models and languages. Note: the lighter shaded diagrams belong in the overall picture but are not currently under consideration.

platforms include the IBM SP2, Cray T3D, Intel Paragon, DEC SMP AlphaServer, and SGI PowerChallenge. The ZPL compiler from the University of Washington supports the Intel Paragon, Cray T3D, IBM SP2, KSR2 and network of DEC Alpha workstations. Such a wide availability allows for detailed comparisons and analysis between HPF and ZPL as well as among HPF compilers.

The reader may question at this point the practical purpose of comparing HPF and ZPL since it appears that HPF, being based on Fortran, is more likely to be accepted as the standard parallel language. My contention, to be borne out by the results of this thesis, is that HPF lacks a robust programming model to assure the programmers that an algorithm implemented in HPF will be scalable and portable. In this respect, the choice of ZPL for a comparison with HPF is appropriate since ZPL has a strong foundation in a programming model. One can argue that because of the high visibility of HPF, a failure of the language due to performance problems will not only disappoint the user community but will also have an adverse effect on continuing research in parallel systems as a whole. Therefore, if HPF is failing to meet user expectations, it is imperative that we understand the source of the problem.

Chapter 3 will focus on key features of both languages that are critical to scalability and portability. While it is difficult to measure the syntactic appeal of a language abstraction, the ease of use ultimately depends on whether the performance of the abstraction meets the user expectation. In Chapter 4 this will be quantified through an experimental characterization of a number of basic language abstractions in both languages.

Chapter 5 will carry the comparison to the NAS benchmarks. This chapter will also briefly consider the interesting issue of native and nonnative compilation.

Chapter 6 will consider the problem of a computation with a recurring dependence. A number of solutions are studied and a new language abstraction is proposed that generalizes the parallel prefix operation.

Finally Chapter 7 will summarize the findings of this thesis and outline future work.

Chapter 2

Programming Models for Data Parallel Problems

2.1 Introduction

In Chapter 1 portability and scalability have been identified as critical requirements for software development, particularly for parallel applications. Portability for parallel programs, however, is hampered by the rich diversity in parallel architectures. Parallel machines in general present one of two memory models: shared memory and nonshared memory. If the programmer is to adhere to the memory model of the machine, then different versions of the program must be written, contradicting the goal of portability. At the same time, choosing one model may preclude executing the program on machines that do not support the model.

To remedy this problem, a straightforward solution is to emulate the program's programming model if such a model is not directly supported by the machine. Since a programming model is an abstraction, it can be readily constructed in software, albeit at a certain cost. For instance, on a nonshared memory machine, a software layer can provide the view of a single address space. Conversely, a nonshared memory programming model can be created trivially on a shared memory machine by emulating the send

and receive. The focus then shifts to the cost of such an emulation because this cost directly affects the scalability. If it is negligible, then the portability problem with respect to the shared and nonshared memory models can be considered solved.

Shared memory programs can be executed on nonshared memory machines by using the Shared Virtual Memory system proposed by Li and Hudak[Li & Hudak 89]. In this approach, the operating system maintains a consistent cache of memory pages to create the view of shared memory in a nonshared memory machine. Priol and Lahjomri [Andre & Priol 92] measured the performance of a number of shared memory programs running on a Shared Virtual Memory system for the iPSC/2 and compared against the native nonshared memory programs on the same machine. They found that the shared memory programs tend to have difficulty with the granularity of sharing.

In this chapter, we examine the case of executing nonshared memory programs on shared memory machines. Real world instances of this approach can readily be found. Large scale nonshared memory machines (e.g., Intel Paragon, IBM SP2) are targeted by data parallel languages such as HPF[Forum 93] and ZPL[Lin & Snyder 93]. At the same time, as a testimony to the low cost and effectiveness of the class of small scale bus-based shared-memory machines, many computer vendors presently offer symmetric multiprocessors (SMP). It is logical then to add the proper software emulation on these SMPs to execute the nonshared memory programs generated by HPF and ZPL. In fact, ZPL programs can be run on both shared memory machines such as the KSR and nonshared memory machines such as the Paragon. Similarly, the DEC HPF compiler supports both a network of workstations and the DEC SMP server [Harris et al. 95].

One motivation for choosing the nonshared memory model is that this model offers accurate performance prediction. Anderson and Snyder [Anderson & Snyder 91] analyzed several algorithms developed with the shared and nonshared memory models and found that the shared memory model produces overly optimistic performance prediction that leads to suboptimal algorithms. To achieve portability for the nonshared memory model requires emulation that may affect the scalability and this tradeoff needs to be

quantified. In this chapter, this question is formulated as a comparison of shared memory and nonshared memory versions of two applications, LU Decomposition and Molecular Dynamics simulation. The comparisons are made in two ways:

1. The data reference pattern of each application is used to construct a simple analytical model of the parallel execution to predict the behavior.

2. The performance of the programs is measured on five shared memory machines with widely differing memory organizations.

In related work, Lin and Snyder [Lin & Snyder 90] have examined the performance of shared and nonshared memory programs on several shared-memory machines. They found that the nonshared memory program can outperform the shared memory version in many cases. However, the study included only two simple programs, matrix multiplication and Jacobi iteration, and only two machines, the Sequent Symmetry and the BBN Butterfly GP1000. Although the results were interesting, the data set was deemed insufficient for making generalizations.

Leblanc [LeBlanc 86] also made a similar comparison using Gaussian Elimination on the BBN Butterfly. He observed that the model should be chosen based on the nature of the application, and that the shared model may encourage too much communication. However, since the version of Gaussian Elimination used did not include the difficult partial pivoting step, the computation is not very different from matrix multiplication. In addition, the measurements were collected from only one older class of machine that uses relatively slow processors.

The study in this chapter contributes to the results from previous work by including many contemporary large scale shared memory machines and by considering an analytical model for the benchmarks that corroborates the observed performance. The second point is particularly significant because measured performance may include unknown system effects that may bias the results. An analytical model that is validated by real data will confirm that the hypothesized effect, i.e., the memory model, is indeed the primary effect.

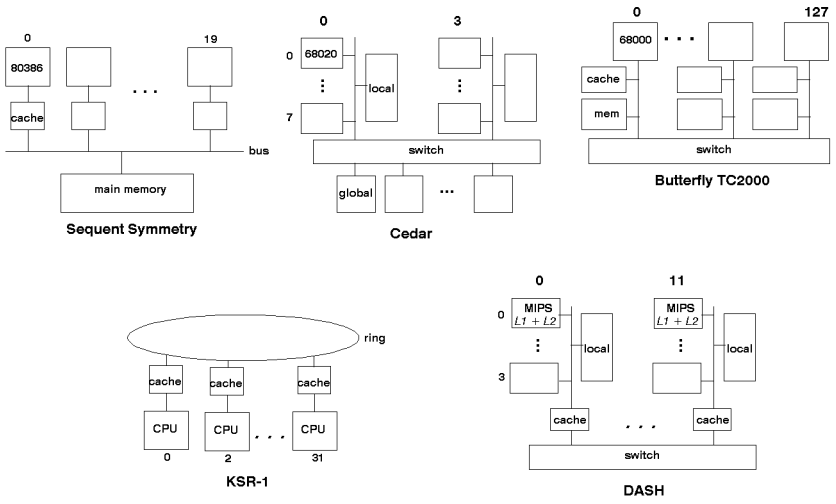


Figure 2.1: Machine memory hierarchy

The remainder of the chapter is organized as follows. Section 2.2 defines the scope of the problem and describes the methodology of the experiments, including the machines and the implementation. Sections 2.3 and 2.4 describe in detail each of the two applications and the results. Conclusions are found in Section 2.5.

2.2 Methodology

To facilitate the discussion in this chapter, we use the subscripts s and ns: P_s is a shared memory program while P_{ns} is a nonshared memory program.

2.2.1 The Machines

The shared-memory machines used in the experiment are the Sequent Symmetry, the BBN Butterfly, the Cedar at CSRD, Illinois, the Kendall Square Research KSR-1, and the DASH at Stanford. They represent a wide range of memory hierarchies from relatively uniform access (Sequent) to non-uniform access (Cedar, Butterfly, KSR-1, and

Table 2.1: Characteristics of the shared memory machines
(Note: (1) access ratio is normalized to the access time of the memory closest to the processor, (2) * is sum of all local memories)

Machine	Sequent	CSRD Cedar	Butterfly	Kendall Square	DASH
Model	Symmetry A	Cedar	TC2000	KSR-1	DASH
Site	U. Washington	U. Illinois	LLNL	U. Washington	Stanford
nodes	20	32	128	32	48
processors	Intel 80286	MC 68020	MC 88100	custom	MIPS R3000
I cache	combined	16 Kb/node	32 Kb/node	256 Kb/node	64Kb/node
D cache	64 Kb/node	128 Kb/cluster	16 Kb/node	256 Kb/node	64Kb/L1
					256Kb/L2
					128 Kb/cluster
local mem	0	32 Mb/cluster	16 Mb/node	32 Mb/node	14 Mb/node
global mem	32 Mb	256 Mb	2 Gb*	1 Gb*	168 Mb*
network	bus-based	omega	butterfly	ring	mesh
access ratio	1	1:4.5	1:3.7:12.7	1:10:75:285	1:4.5:8:26

DASH). The Sequent, KSR-1 and DASH employ hardware coherent caches while the Cedar and the Butterfly do not. Figure 2.1 and Table 2.1 show the general organization and some relevant characteristics of each machine. Note that the access ratio is the access time of each memory level normalized to the access time of the level closest to the processor. This parameter is intended to give an indication of the depth of the memory hierarchy.

One important point worth noting is that although several machines are being studied, the focus of the comparisons is between two programs on each machine. Therefore, while the speed of the processor relative to the memory performance plays a central role in determining the performance of a program, it can be factored out in a comparison between programs on the same machine. The same argument applies to the differences in the memory performance between the machines due to the memory architecture, e.g., memory bandwidth, cache line size, etc.

Following is a brief description for each parallel machine used in this study.

The Sequent Symmetry is a small scale bus-based machine. Because of the low speed of the processors, the bus is able to support 20 nodes. Cache coherency is maintained by bus-snooping using a modified Illinois protocol[Gifford 87]. Since the main memory

resides on the bus, the access time is the same from any processor cache, and since there is no control over the cache, each processor will only see a certain average access time. Because of these characteristics, the Sequent bears the closest resemblance to the PRAM model[1] when compared to the other machines. For this reason, the access ratio of the Sequent in Table 2.1 is approximated as 1.

The Cedar has a cluster architecture: each bus-based cluster contains 8 processors sharing a local memory and the clusters are connected to the global memories through a switch. Therefore there are two levels of memory hierarchy. Note that the intra-cluster communication can be done at the cluster memory level, but inter-cluster communication must use the global memory.

In the Butterfly TC2000, each processor node contains a cache and a local memory. Each node is connected to the local memories of all other nodes through a multistage interconnection network (MIN); the global memory thus consists of the aggregate of all local memories. There are three levels of memory hierarchy: the cache, the local memory, and the remote memory. Although memory locations can be assigned a variety of cache attributes, the most commonly used attributes are cacheable local and non-cacheable global, which represent only two levels of memory hierarchy. Since there is no hardware coherency mechanism, the machine provides various cache invalidation functions to support software caching. Finally, to avoid hot spots in this memory organization, the global address space is interleaved across all nodes.

The KSR-1 employs an AllCache architecture [Kendall Square Research 92]: instead of a main memory, each node possesses a large cache that is kept coherent with all other caches through a snoopy mechanism at the ring level and a directory-based scheme within the ring hierarchy. In addition to the main cache, there are also instruction and data subcaches on each node. The nodes are connected in a hierarchy of rings; therefore, there are multiple levels of memory hierarchy, beginning with the subcache, to the local cache, the caches within the same ring, the caches within the next ring level, and so on. Because

[1] Parallel Random Access Memory model: each processor can access any memory location in unit time.

the unit of communication on the ring is a 128 byte cache line, the granularity of shared data is an important issue. While the architecture provides a combining mechanism by servicing a cache miss at the lowest level of the ring hierarchy, the machine used in the experiment only contains one ring; therefore this combining effect is not visible.

The Stanford DASH also uses a directory scheme for cache coherency [Lenoski et al. 93]; however, it is organized as a collection of bus-based clusters connected in a mesh topology. Each cluster contains 4 processor nodes, a cluster memory and a cluster cache. Each node in turn contains a level 1 and a level 2 cache, with coherency maintained at the level 2 cache through bus snooping. Consequently, there are at least 5 levels of memory hierarchy: level 1 cache, level 2 cache, cluster memory, remote cluster memory, and remote dirty cluster memory. The cluster cache that resides in the interface between a cluster and the other clusters combines requests to the same address. Since there is a home node for each memory location, the system also provides functions for data placement.

2.2.2 The applications and the implementations

In this study, we focus on the class of data parallel applications by using the LU decomposition problem and the WATER benchmark from the Stanford SPLASH suite. Although there are other important classes of parallel applications, the data parallel class covers a large number of scientific problems and this class stands to benefit immediately and significantly from parallelization.

Efficient algorithms exist for both problems in both shared and nonshared memory models, and the algorithms employ static data partitioning and load balancing. The following sections will analyze these algorithms in detail. Note that because our focus is on the portability issue of parallel programs, we are not considering applications with an irregular structure for which an efficient nonshared memory solution may not be obvious. While an algorithm for such irregular applications may perform reasonably well on a shared memory machine, the same algorithm is unlikely to perform well on

a nonshared memory machine with emulated shared memory because of the significant difference in the memory latency. In other words, with respect to portability, we consider that portable solutions for such problems have not yet been developed.

In implementing the algorithms, all programs are written in the SPMD style. The performance is measured only for the useful computation; the initialization costs are not included.

In P_s, data resides in the global memory and the standard lock and barrier mechanisms are used for synchronization. In instances where an architecture provides a major feature to support the shared memory, simple machine specific optimizations for data placement are used, and they are described in the following sections. In general, however, aggressive optimizations that are not portable are excluded.

In P_{ns}, data is placed in the level of memory closest to the processor. On machines with coherent caches, this effect is achieved automatically through the data reference pattern: since a processor only accesses its data partition, data is allowed to migrate to the highest level of the cache. Note that although data in P_s also migrates to the level closest to a processor, the data may be written by other processors and as a result be forced out of this level. In general, data locality (in the processor dimension) is an inherent advantage of P_{ns}.

The sends and receives are emulated with straightforward block copy, and simple one-reader/one-writer ports implement some connection topology (binary tree, mesh, etc.). The buffers used for communication are placed in the global memory. Data communication is more expensive in P_{ns} than in P_s because it must be emulated through software, incurring costs in code to manage the buffers as well as additional loads and stores to read and write the buffers. In contrast, P_s communicates data simply by writing to the shared address space. Although the software emulation can be further optimized by passing pointers, such optimizations are excluded in this study.

Given the advantages and disadvantages described above, the performance tradeoff for P_{ns} will be between the communication overhead and better processor locality.

An important issue in implementing the shared and nonshared memory versions is the relative ease of the implementations. Given the current state of the art in parallel languages, a shared memory program is generally easier to write than a nonshared memory program and this holds true in our implementations. However, since our focus is portability, there are several points worth noting. First, it is easier to arrive at a working shared memory program from scratch because the model allows the programmer to ignore data placement. In many cases, the pattern of reference allows the data to migrate naturally to the processors that use them, yielding good performance with little effort by the user. However, it is generally recognized that the program must be optimized for data locality to obtain scalable performance. This optimization step is difficult since machine specific techniques are not portable and portable techniques will likely require significant restructuring of the program or the algorithm. Second, the difficulty in writing a nonshared memory program reflects the lack of appropriate language support for the nonshared memory model rather than a fundamental limit in the model. Current developments in parallel languages for nonshared memory machines such as HPF and ZPL may provide a solution to this problem. As a result, we believe the issue of ease of programming should be excluded for the focus of this chapter. This issue will be revisited in the following chapters.

2.3 LU Decomposition

The following two sections describe the two sets of experiments. The problem and the algorithm are first described, followed by an analysis of the data locality in the P_s and P_{ns} versions. Results and discussion follow.

2.3.1 The problem

Typical of matrix problems, LU decomposition is a numerical method for solving large systems of linear equations. Given the system of equations:

$$Ax = b$$

where A is the coefficient matrix and b is the constant vector, A is decomposed into a lower and upper triangular matrices L and U:

$$LUx = b$$

Letting Ux = y, we can solve directly for y by forward-elimination

$$Ly = b$$

and then for x by backward-substitution.

$$Ux = y$$

The sequential algorithm consists of iterating over the diagonal of the matrix, each iteration consisting of two steps: partial pivoting (line 2) and row update (lines 3:6).

LU_{seq}

```
(1)     for (k=0; k<row; k++)
(2)         partial pivot
(3)         for (i=k+1; i<row; i++)
(4)             A_ik = A_ik/A_kk;
(5)             for (j=k+1; j<col; j++)
(6)                 A_ij = A_ij - A_ik * A_kj
```

The partial pivoting step is necessary for numerical stability in the division step due to the limited precision in digital computers: it involves searching the pivot column k for the largest element and then swapping that row with the pivot row k. With a complexity

of $O(n^2)$ compared to the $O(n^3)$ of the row update step, this step only constitutes a minor part of the computation; however, it introduces additional serialization into the algorithm.

2.3.2 The parallel algorithms

Since LU decomposition is a well-studied problem, optimized parallel algorithms are widely available in the literature [Ashcraft 91, Karp 87, Robert 90].

The computations in the row update step for each iteration are independent and can be parallelized easily. The partial pivoting step is more difficult to parallelize effectively because the parallelism available is small compared to the communication/synchronization required for parallelization. Therefore, the optimized algorithm employs pipelining. During the current iteration, a processor performs the complete LU decomposition on a set of t columns and saves the transformation, while the remaining processors update the submatrix using the saved transformation from the previous iteration. The value of t controls the granularity of the task partitioning and is chosen to best balance the workload and the communication/synchronization overhead.

The pseudo-code for the optimized parallel LU decomposition algorithm is shown below.

```
        LU_s
(1)       P0 factors col[0:r]
(2)       for (k=r; k<row; k+=t)
(3)          switch transformation buffer
(4)          if (own_column(k))
(5)             updates col[k:k+t] using transformation (k-t)
(6)             factors col[k:k+t] saving transformation k
(7)          else
(8)             updates col[k+t:col-1] using transformation (k-t)


        LU_ns
```

```
(1)     P0 factors col[0:t]
(2)     and broadcasts the transformation
(3)     for (k=r; k<row; k+=t)
(4)       if (own_column(k))
(5)         updates col[k:k+t] using transformation (k-t)
(6)         factors col[k:k+t] saving transformation k
(7)         broadcasts the transformation k
(8)       else
(9)         updates col[k+t:col-1] using transformation (k-t)
```

LU_s and LU_{ns} thus implement the same algorithm. The differences are:

1. Any processors can update any portion of the matrix in LU_s, while the matrix is partitioned statically by columns in LU_{ns}. Because the iteration traverses the diagonal of the matrix, partitioning the columns by blocks will result in a poor load balance in LU_{ns}. Some processors will be idle once k has passed their sections. To alleviate this problem, sets of r columns are assigned to the processor in an interleaved fashion (cyclic).

2. LU_s requires barrier synchronizations before and after switching the transformation buffer in line 3. LU_{ns} requires broadcasting the newly computed transformation buffer to all processors in lines 2 and 7.

On the Cedar and the Butterfly where there exists a local memory but no coherent cache, we improve the data locality of LU_s by performing software caching in the innermost loop. On Cedar where there is a cluster level memory, we also optimize the communication in LU_{ns} by using the cluster memory for intra-cluster messages and the global memory for inter-cluster messages. Other techniques for improving LU by using primitives at the user level have been proposed by Qin[Qin & Baer 97].

2.3.3 Volume of data references

Since LU decomposition is a static algorithm, the amount of scalar computation and the volume of references to the matrix should be nearly identical for the sequential, shared memory, and nonshared-memory versions. [2]. The total number of processor cycles to compute and perform this volume of references represents a lower bound on the execution time of the program. Since the data placement is static in the memory hierarchy and the ratio of access times to each level of memory is known, we can derive an indicator of the relative performance of P_s and P_{ns}. We assume the synchronization cost is small and the load balance is perfect. For the following analysis, we only assume a local and global level of memory and that read and write have the same latency. The matrix is also assumed to be square, i.e., n=column=row.

Referring to the LU_{seq} algorithm above, in each outermost iteration k the partial pivot step consists of scanning a column (1 read) and swapping two rows (2 reads + 2 writes) beginning from the diagonal element. The number of references is:

$$(3r + 2w) * (n - k)$$

The row update step consists of dividing the column by the pivot element (2 reads + 1 write) and adjusting the rows (3 reads + 1 write). The number of references is:

$$((2r + 1w) + (3r + 1w) * (n - k - 1)) * (n - k - 1)$$

Summing up over the diagonal iterations, we obtain the volume of references to the matrix:

$$\sum_{k=0}^{row-1} (3r + 2w) * (n - k) + ((2r + 1w) + (3r + 1w) * (n - k - 1)) * (n - k - 1)$$

Simplifying and substituting the summations with the equivalent polynomials, we obtain the expression:

$$n^3(r + \frac{1}{3}w) + n^2(r + w) + n(r + \frac{2}{3}w)$$

[2] Discounting small variations due to variable reuse and the references of global parameters.

In addition to accessing the matrix data, the transformation data in P_{ns} needs to be broadcast to the worker processors. This is done through a binary tree: the buffer is sent to the root processor and is propagated down the tree. Each message requires a pair of sends and receives; each send and receive operation in turn involves a local read and a global write, or a global read and a local write, respectively. A message transmission then requires ((2 global + 2 local) * size) references. The elapsed time for propagating through the binary tree requires the equivalence of $(\log p + 1)$ transmissions. The size of the transformation buffer in each k iteration is (n-k)*t, where t is the parameter controlling the task granularity. The total time for all tree broadcasts is then:

$$(\log p + 1) * (2global + 2local) * \sum_{k=0, k=k+t}^{n-1} (n - k) * t$$

Substituting the summation and simplifying yields:

$$(\log p + 1) * (global + local) * n * (n + t)$$

We can compute an estimate of the relative performance of P_s and P_{ns} with the assumptions:

1. The matrix references for P_s will be to the global memory and for P_{ns} to the local memory.

2. The computation and thus the matrix references are perfectly distributed among the processors.

3. P_{ns} requires the additional tree broadcast operations.

4. The references are free of contention.

Table 2.2 summarizes the expressions for the reference counts and the floating point operation counts (FLOPS) in part (a); part (b) shows the reference counts that represent the elapsed time for the tree broadcast; and part (c) tabulates the reference counts to the local and global memory based on the assumption that data is placed in global memory

Table 2.2: Volume of data references for LU

Phases	FLOPS	read	write
partial pivot	$\frac{1}{2}(n^2-n)$	$\frac{3}{2}(n^2+n)$	n^2+n
column update	$\frac{1}{6}(4n^3-3n^2-n)$	$\frac{1}{2}(2n^3-n^2-n)$	$\frac{1}{3}(n^3-n)$
Total	$\frac{2}{3}(n^3-n)$	n^3+n^2+n	$\frac{1}{3}(n^3+3n^2+2n)$

(a) Flops and reference count for LU computation phases: n = size of nxn matrix

Phases	read	write
Total	$\frac{1}{2}n(n+r)(\log p+1)(loc+glob)$	$\frac{1}{2}n(n+r)(\log p+1)(loc+glol)$

(b) Elapsed time for LU_{ns} communication in terms of reference count: n = size of nxn matrix, p = processor number in powers of 2

Program	local read+write	global read+write
LU_s	0	$\frac{1}{3}(4n^3+6n^2+5n)$
LU_{ns}	$\frac{1}{3}(4n^3+6n^2+5n)$ $+n(n+r)(\log p+1)$	$n(n+r)(\log p+1)$

(c) LU_s and LU_{ns} references to memory hierarchy

in LU_s and in local memory in LU_{ns}. From this information, we can derive a simple model for the parallel execution of LU_s and LU_{ns}:

$$total\_cycles = communication\_cost + parallel\_task$$

$$parallel\_task = \frac{1}{p}(FLOPS * FLOPS\_cycle + global * global\_cycle + local * local\_cycle)$$

Figure 2.2 plots the speedup based on the number of cycles required by P_s and P_{ns} for ratios of memory hierarchy of 1:1, 1:2 and 1:4, with n=512, t=4, and FLOPS_cycle=1.

Our simple model predicts that P_{ns} easily outperforms P_s when there is any gap between the local and global memory. Naturally, many factors are ignored, such as the load balancing, the synchronization, the network contention, the actual higher cost for emulating the communication, etc. However, if the data locality controls the first order effects and these factors are secondary, then the estimate can be qualitatively correct. In the next subsection we will look at the results measured on the five machines.

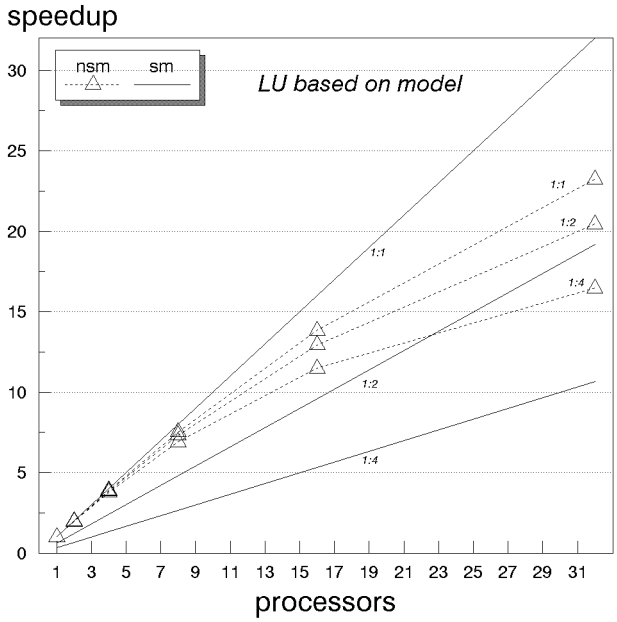


Figure 2.2: LU speedup based on model: n=512, t=4

2.3.4 Performance results

Figures 2.3 through 2.8 show the performance and the speedup of the two versions of LU decomposition on the five machines for three different problem sizes. The speedups are based on the performance of a straightforward sequential version of LU decomposition.

Referring to the predicted speedup curves in Figure 2.2 and the local:global access ratio for each machine in Table 2.1, we find that the results match the model prediction well.

We first consider the results from the Sequent, the Cedar and the Butterfly since the ratios of these machines are more precise. The ratio of 1:1 on the Sequent gives LU_s a slightly better performance than LU_{ns} for all problem sizes. On the Cedar where the ratio is 1:4.5, LU_{ns} offers the better performance. Although the Butterfly has three levels of memory hierarchy, the program only uses two levels (the default cacheable local and noncacheable global attributes); therefore the effective ratio is 1:12.7. This large gap in access latency translates directly into a large gap in the performance between

LU_s and LU_{ns}. The steep hierarchy on the Butterfly gives LU_{ns} an advantage that far outweighs the nonshared memory simulation overhead. We note a number of program optimizations on the Cedar and the Butterfly:

1. For LU_s on Cedar and Butterfly, software caching is performed in the innermost loop by copying a column into local memory for performing the column update; this prevents repeated access of the same column from global memory while updating using the saved transformation.

2. For LU_{ns} on Cedar, intra-cluster communication uses the cluster memory, while inter-cluster communication uses the global memory.

Since the Butterfly cache is controlled by software, more advanced caching techniques may improve the performance of LU_s, but it seems difficult to recapture the large performance gap. It thus appears that a nonshared memory program matches well a large scale shared memory machine with *private* per processor cache. Since the Sequent has a coherent cache while the Butterfly does not, a natural question is whether the behavior found on the Sequent would be observed on the Butterfly if a coherent cache were implemented. A coherent cache improves the performance of both P_s and P_{ns}: it will reduce the global data references in P_s and effectively eliminate the spin-lock traffic in P_{ns} communication.

To search for an answer, we consider the results from the KSR-1 and the DASH, two machines that employ large scale coherent caches. Given the very large ratios for these machines, our simple model predicts that the shared-memory version would not yield any speedup. On the other hand, if we extrapolate from the performance on the Sequent, the coherent cache would be able to hide the memory hierarchy and to present a more uniform access to memory, thus giving LU_s a slight advantage over LU_{ns}. The results show that LU_s achieves a reasonable speedup on both machines, but that it trails LU_{ns} by a significant amount in both actual performance and speedup. Clearly, the actual behavior lies in the middle ground between our two extreme predictions: by reducing

the number of remote accesses, the coherent cache is very effective in reducing the gap in the memory hierarchy, but not to the degree where data locality is rendered unnecessary. The difference in fetching from local and remote memory by the cache is visible in the program performance.

With respect to scaling with problem size, the relative difference in performance between LU_s and LU_{ns} is maintained in every case, and the speedup improves as the problem size increases. Not surprisingly, this behavior reflects the dominant computation cost relative to the cost of communication or memory references, $O(n^3)$ versus $O(n^2)$ for LU.

With respect to scaling with the number of processors, we observe that while the speedup and performance appear reasonable for up to 32 processors on Cedar, KSR-1 and DASH, neither LU_s nor LU_{ns} achieves any speedup beyond 32 processors on the Butterfly when the number of processors approaches 100. A partial explanation for the poor scaling in LU_{ns} can be found in our simple implementation of the communication: (1) all messages are point to point, and (2) a processor spin-locks on a global variable while waiting for an empty write port or a full read port. Each of these factors constitutes a component in the overall communication cost that increases with the number of processors.

2.4 Molecular Dynamics Simulation

2.4.1 The problem

The WATER benchmark from the Stanford SPLASH suite is a simulation of several hundred water molecules in a cubical box in the liquid state at room temperature [Singh et al. 92]. The program is representative of the n-body problem, in which each body interacts in certain ways with all other bodies in the system. In this case, the simulation computes the forces and potentials among the water molecules to predict various static and dynamic properties of water. To compute all pair wise interactions, a processor

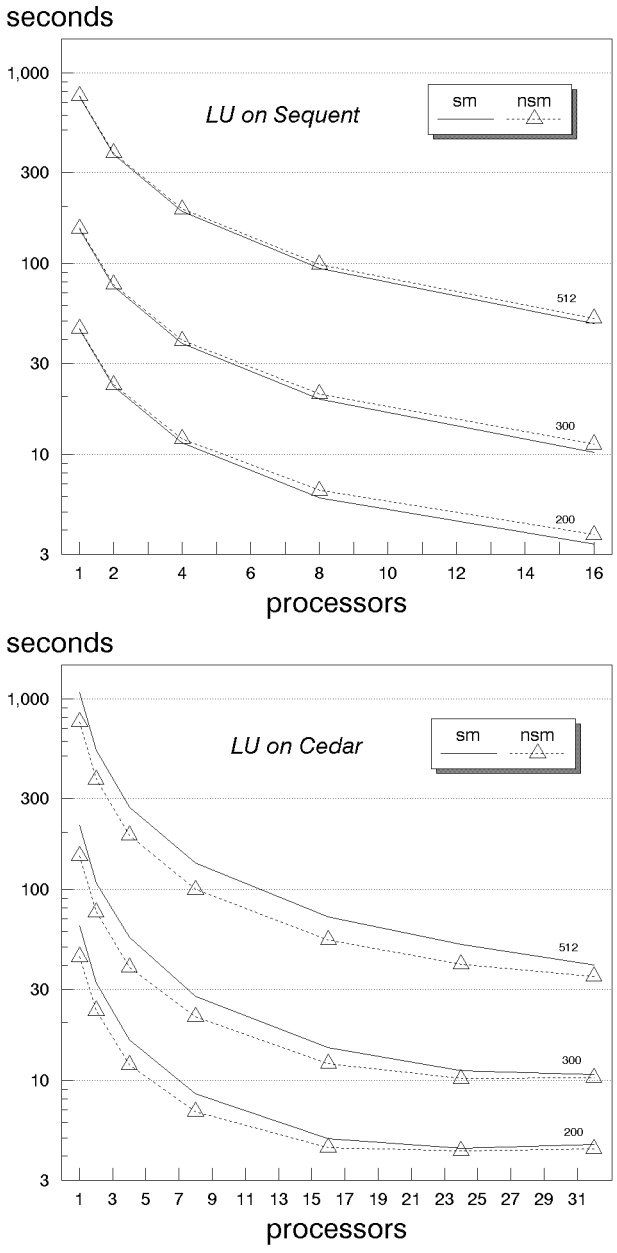


Figure 2.3: LU execution time: Sequent and Cedar

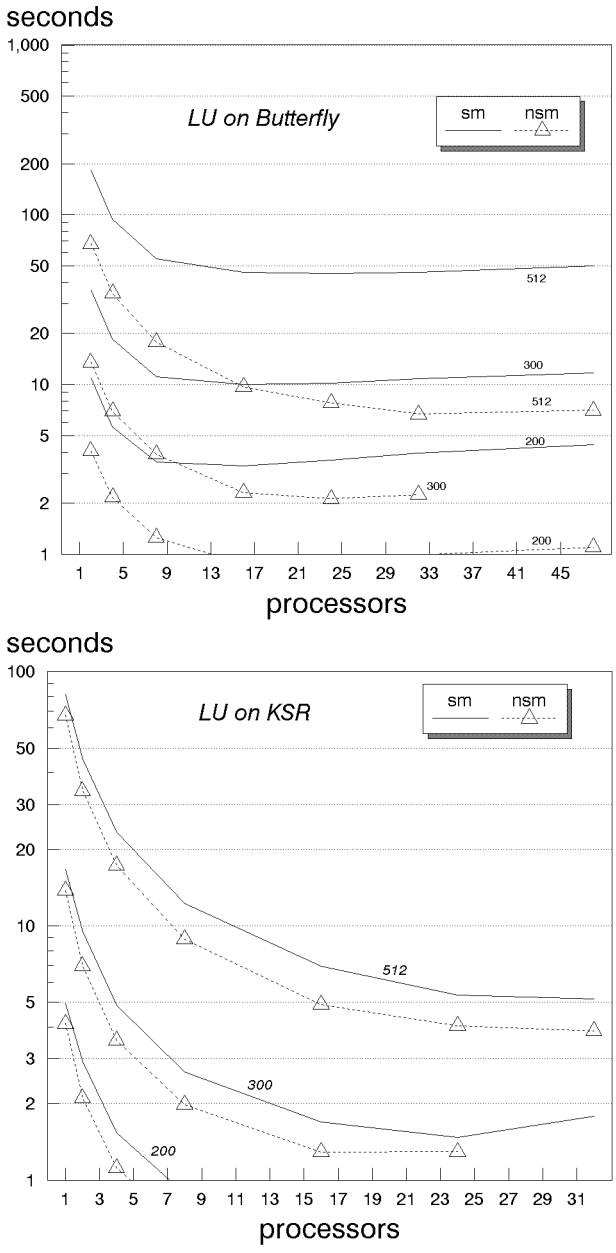


Figure 2.4: LU execution time: Butterfly and KSR

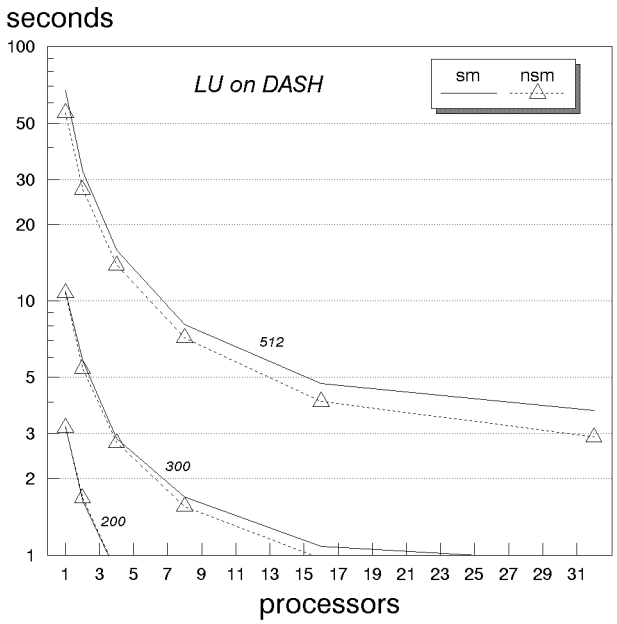


Figure 2.5: LU execution time: DASH

in any partitioning scheme will have to access the data in all sections, giving an initial appearance of poor data locality in the problem. However, the version in this study achieves a good speedup thanks to a favorable ratio of computation to communication, as will be described in the following subsections.

2.4.2 The algorithm

The program was manually parallelized from a sequential version, the MDG benchmark in the Perfect Club suite. After initializing the displacements and velocities, the algorithm consists of iterating over a large number of time steps until the system converges to a steady state. Each time step consists of seven computation phases separated by barrier synchronizations:

1. Predict new values for displacement and the derivatives.

2. Compute the intramolecular forces between the atoms of each molecule.

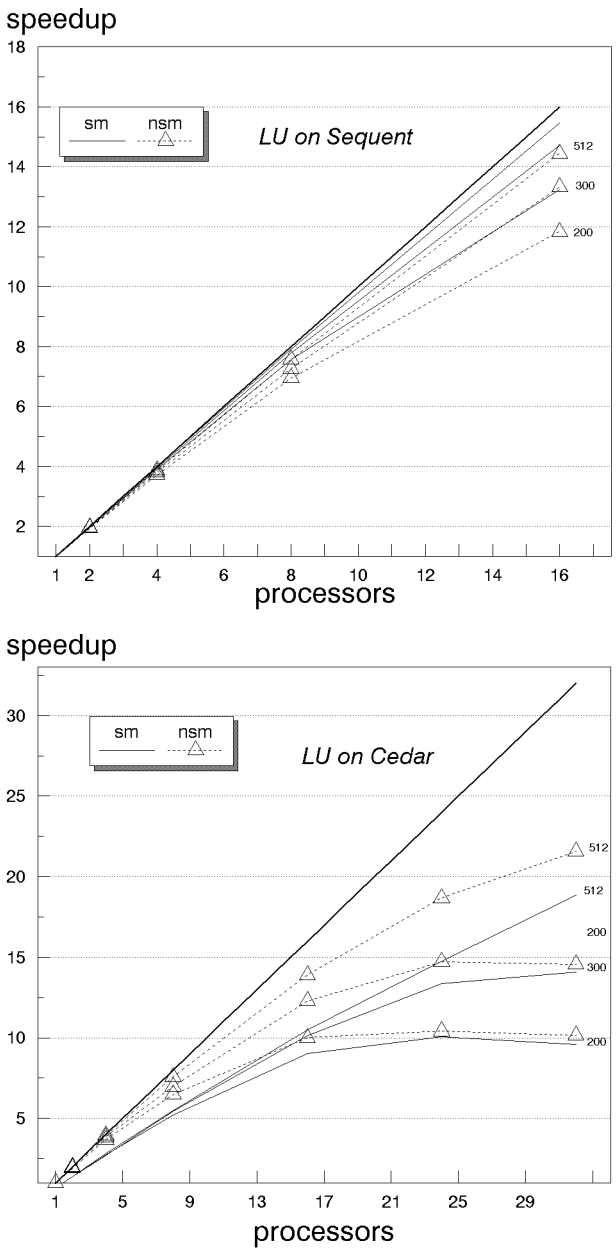


Figure 2.6: LU speedup: Sequent and Cedar

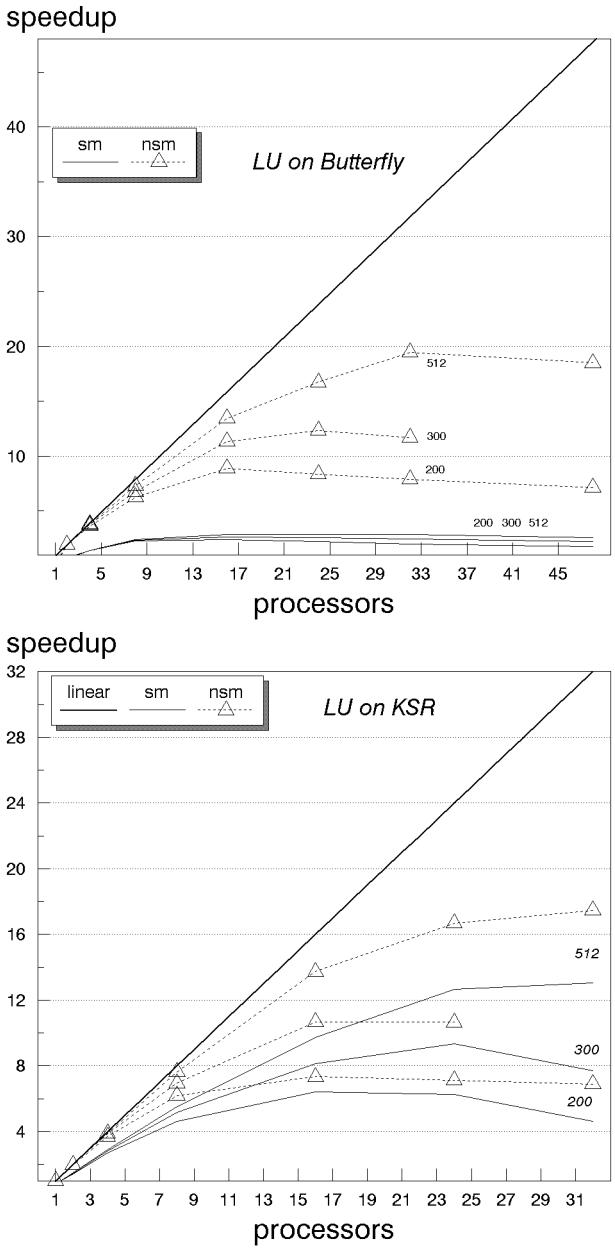


Figure 2.7: LU speedup: Butterfly and KSR

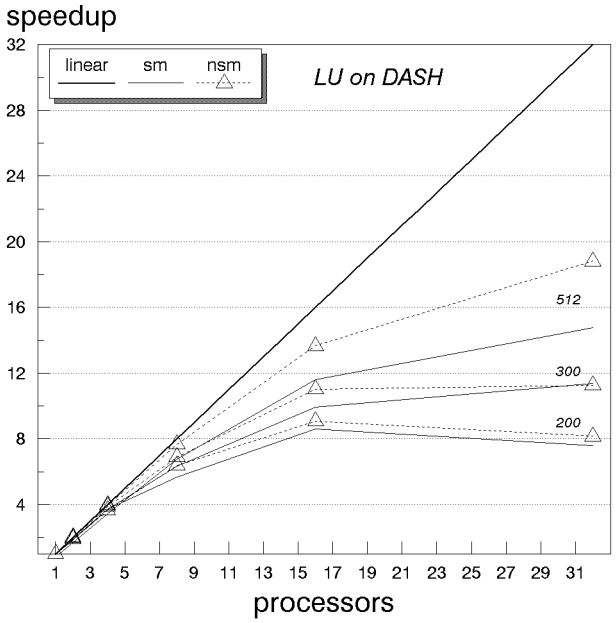


Figure 2.8: LU speedup: DASH

3. Compute the intermolecular forces between the atoms of each pair of molecules.

4. Correct the predicted values for forces.

5. Handle the boundary conditions by moving the molecules back into the box if they are out of the box.

6. Compute the kinetic energy in each of the three spatial dimensions.

7. Compute the potential energy as the sum of the intermolecular and intramolecular potentials.

The computation complexity is $O(n^2)$, but the actual number of pair wise interactions to be computed is reduced by defining a cutoff radius of half the box dimension.

The $WATER_s$ version was ported to the various machines strictly by substituting the parallel macros for lock and barrier synchronization.

$WATER_{ns}$ is derived from $WATER_s$ by replacing the shared data structures with distributed structures and performing a global update at each point where (1) each

partition has to access all other partitions to compute the pair wise interactions, or (2) a global sum has to be computed and broadcasted to all partitions. Communicating through a ring topology, each process computes the interactions within its partition, then sends a copy on a complete trip around the ring; as a partition is received, the process updates both its partition and the traveling partition. When the modified partition returns to its source, it is merged into the original partition. Computing the global sum is done similarly by sending the partial sum around the complete ring.

In both versions, the processor workload is statically assigned and load balancing is not considered a problem due to the uniform distribution of the input data. $WATER_s$ and $WATER_{ns}$ thus perform the identical computation; the only differences are in the placement of the data and the resulting communication.

The algorithm has a computation complexity of $O(n^2)$ and a communication complexity of $O(n)$, which will be described in more detail in the next subsection.

2.4.3 Volume of data references

As with the LU experiment, we create a simple model of the parallel execution of WATER based on the FLOPS and data reference counts.

The computation phases are listed in order in Table 2.3 part (a) together with the count of floating point ops, reads and writes for each time step; the counts are obtained manually from the program. To account for the cutoff range of half the box length, the molecule distribution is assumed to be uniform and those counts that are dependent on the range are divided in half. Part (b) shows the elapsed time of the ring communication in $WATER_{ns}$ in terms of reference counts; note that only 4 phases actually require communication. In part (c), the reference counts to local and global memory are tabulated based on the assumption that data is placed in global memory for $WATER_s$ and in local memory for $WATER_{ns}$. The counts include those references for emulating the communication in $WATER_{ns}$.

As with the LU experiment, a simple model can be derived as:

Table 2.3: Volume of data references for WATER

Phases	FLOPS	read	write
predict val	432n	243n	54n
intra force	42n+223	24n	3n+3
inter force	$163n^2 + 9n$	$46n^2 + 9n$	$4n^2 + 9n$
correct val	135n	81n	63n
boundary	9n	9n	9n
kinetic	24n+3	18n + 3	3
potential	$122n^2 + 128n + 3$	$42n^2 + 33n + 3$	3n+3
Total	$285n^2 + 779n + 229$	$88n^2 + 417n + 6$	$4n^2 + 141n + 9$

(a) WATER computation phases: n = number of molecules

Phases	read	write
intra force	3 (loc+glob)	3 (loc+glob)
inter force	(84n+3) (loc+glob)	(84n+3) (loc+glob)
kinetic	3 (loc+glob)	3 (loc+glob)
potential	(84n+3) (loc+glob)	(84n+3) (loc+glob)
Total	(168n+12) (loc+glob)	(168n+12) (loc+glob)

(b) Elapsed time for $WATER_{ns}$ ring communication:
n = number of molecules; loc, glob = local, global access

Program	local read+write	global read+write
$WATER_s$	0	$92n^2 + 558n + 15$
$WATER_{ns}$	$92n^2 + 894n + 39$	336n+24

(c) $WATER_s$ and $WATER_{ns}$ reference to memory hierarchy

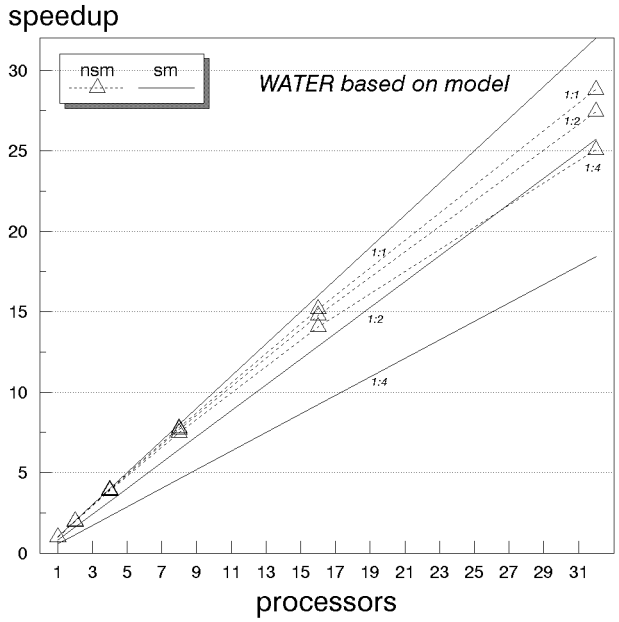


Figure 2.9: WATER speedup based on model

$$total\_cycle = communication\_cost + parallel\_task$$

$$parallel\_task = \frac{1}{p}(FLOPS * FLOPS\_cycle + global * global\_cycle + local * local\_cycle)$$

Figure 2.9 shows the speedup for $WATER_s$ and $WATER_{ns}$ based on the number of cycles required; several ratios of local to global are shown, while the cycle per FLOPS is set to 1. The model predicts that for a local:global ratio of 1:1, $WATER_s$ has the better speedup, but as the ratio increases, $WATER_s$'s speedup degrades quickly and falls below $WATER_{ns}$. Note also that in general the speedups for both versions are better than those for LU. The reason is evident in the large FLOPS count relative to the reference count, indicating a more abundant amount of parallelism.

2.4.4 Performance results

Figures 2.10 through 2.15 show the performance of the two versions of WATER on the five machines for three different problem sizes. The speedup is based on the better uniprocessor performances of the two versions.

As evident in Table 2.3, WATER differs from LU in that the computation dominates the memory accesses and the memory reads dominate the memory writes, although all have the same asymptotic complexity.

On the Sequent where the local:global ratio is 1:1, $WATER_s$ has a near linear speedup while $WATER_{ns}$ is only slightly behind. On Cedar, both versions give virtually the same performance and speedup, although the model predicts that $WATER_{ns}$ would be faster. It is possible that there are some other factors involved that are not included in the model.

On the Butterfly, neither version achieves a speedup beyond 20. Indeed, $WATER_{ns}$'s performance degrades below $WATER_s$ when the number of processors approaches 100. This behavior is consistent with our model although a graph for this configuration was not shown. As the number of processors increases, the parallel computation decreases while the communication remains constant. Therefore, the communication cost in $WATER_{ns}$ will eventually dominate the benefit of the local memory.

On the KSR-1, the performance and speedup of both versions are high and nearly identical. On the DASH, the results of WATER actually differ from that of LU. Although both shared and nonshared-memory versions have the same performance for small number of processors, $WATER_{ns}$ has the worst performance when the number of processors is large.

Model prediction for the DASH and KSR-1 is uncertain because of the combination of the cache and the steep memory hierarchy behind the cache. The effectiveness of the cache depends on the characteristics of the application and the overhead for maintaining consistency. In this case, the high ratio of read to write access implies that the degree of data sharing is relatively small and that the cache hit rate is high. This would

decrease the significance of the memory hierarchy, allowing the cache to present a more effective model of uniform shared-memory. In other words, given the memory access characteristics of the WATER program, the DASH achieves an effective local:global ratio of 1:1 while the KSR-1 gives a ratio that is only slightly worse.

2.5 Conclusion

In this chapter we consider the tradeoff between portability and scalability. The situation arises because two memory models exist at the machine level. Portability limits the choice to one model; therefore on machines that do not directly support this model, it must be emulated by some runtime software. Since scalability may be degraded by the emulation, it is necessary to quantify this cost. Examples for this emulation exist for both choices of model, but we are particularly interested in the nonshared memory model since it more accurately reflects the physical characteristics of a large class of parallel machines.

The question is formulated as a comparison of the shared and nonshared memory implementations of two applications on five widely differing shared memory machines. The performance difference is expressed as a simple factor $\frac{P_{ns}}{P_s}$. Figure 2.16 shows a scatter plot of this factor for each application. Note that the nonshared memory program has the better performance for the points below 1. The solid line is a least mean square curve fit of the data points.

The results show a marked trend that supports our hypothesis. For shared memory machines with a non-uniform memory access time, programs written using the nonshared memory model benefit from being able to better exploit the local memory. As a result, a nonshared memory program tends to be more scalable than a shared memory program on shared memory machines despite the emulation cost. The advantage of the nonshared program is proportional to the *effective gap* between the global and local memory that results from the combined characteristics of the program and the machine architecture.

In this study, LU proves to be more demanding in its data reference pattern (lower read/write ratio); therefore its performance accentuates the effects of the machine archi-

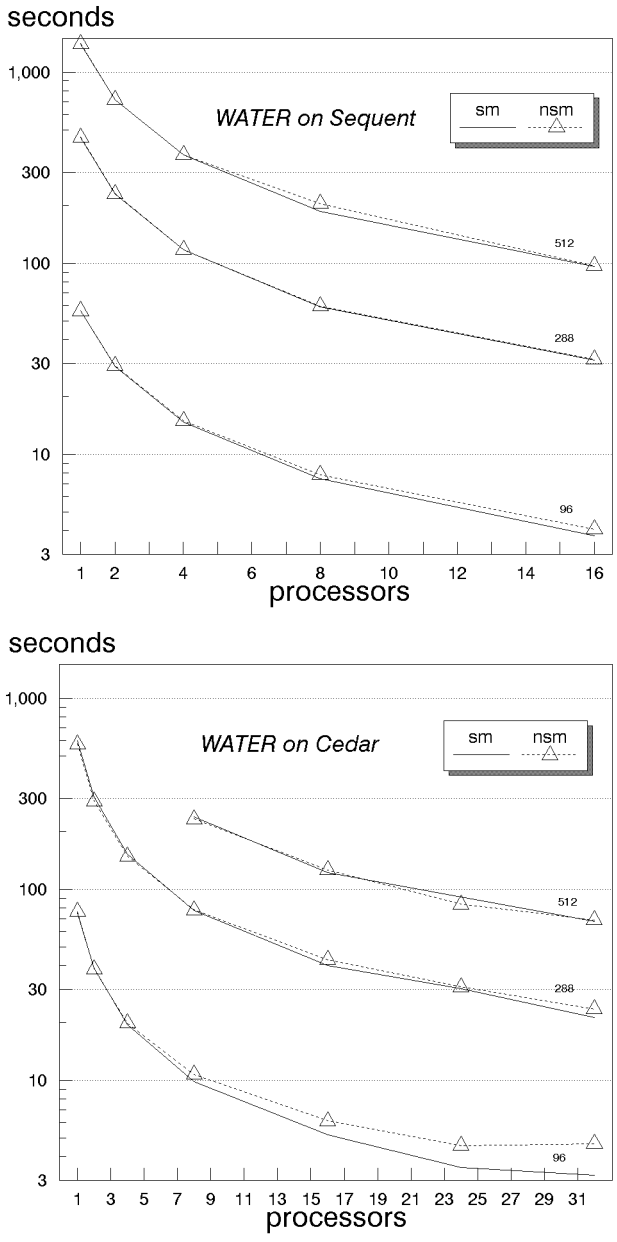


Figure 2.10: WATER execution time: Sequent and Cedar

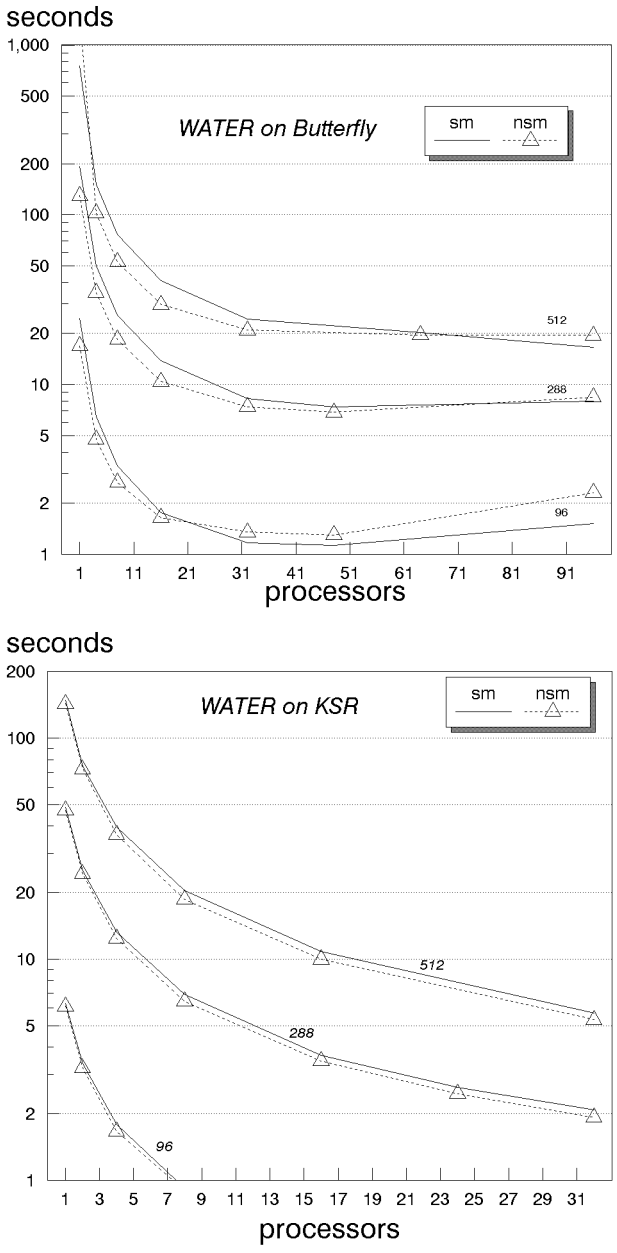


Figure 2.11: WATER execution time: Butterfly and KSR

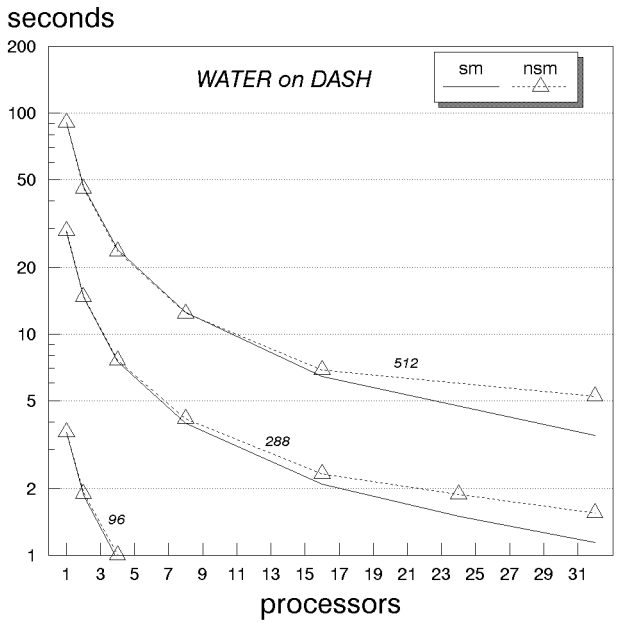


Figure 2.12: WATER execution time: DASH

tecture. We found that the *effective gap* is significant when the ratio between the global and local memory latency is large, e.g., the Butterfly, but it is otherwise small if the ratio is small or nonexistent, e.g., the Sequent. Although the KSR and DASH have a deep memory hierarchy with large differences in latency between the memory levels, the hardware coherent cache plays a significant role in reducing the *number* of long latency accesses and thus the *effective gap* in the memory hierarchy. The benefit of the coherent cache is clear in WATER, which has a data reference pattern favorable for caching (higher read/write ratio). However, when we compare LU and WATER, it is evident that the effectiveness of the coherent cache is contingent on the characteristics of the application. Cast in the modeling framework described in Chapter 1, WATER's behavior fits the model that the cache is designed for, while LU's behavior is less cooperative.

Chapter 1 identified three requirements: portability, scalability, and ease of use. In this chapter, we have largely ignored the last component. The shared memory model is generally considered convenient to the users while the nonshared model is not. On the

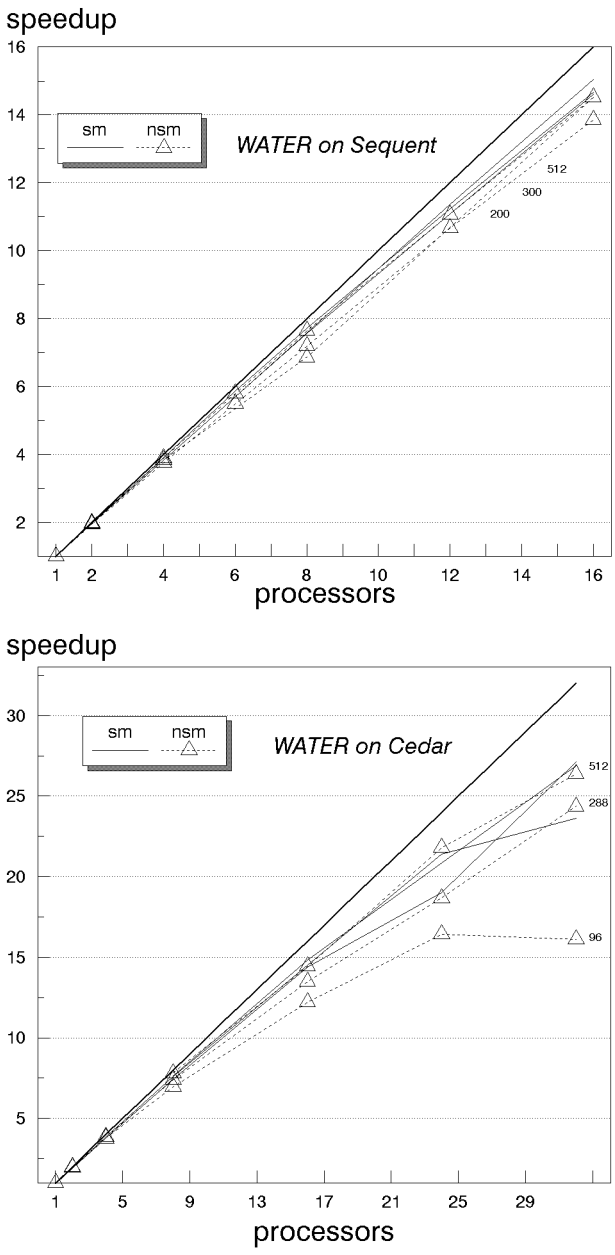


Figure 2.13: WATER speedup: Sequent and Cedar

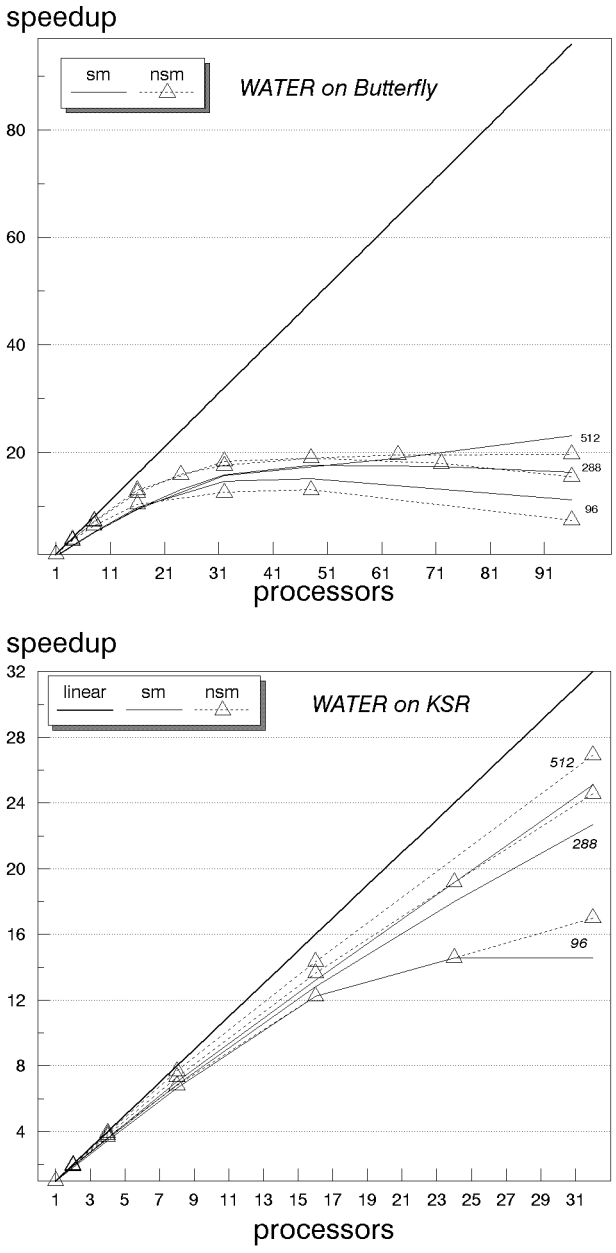


Figure 2.14: WATER speedup: Butterfly and KSR

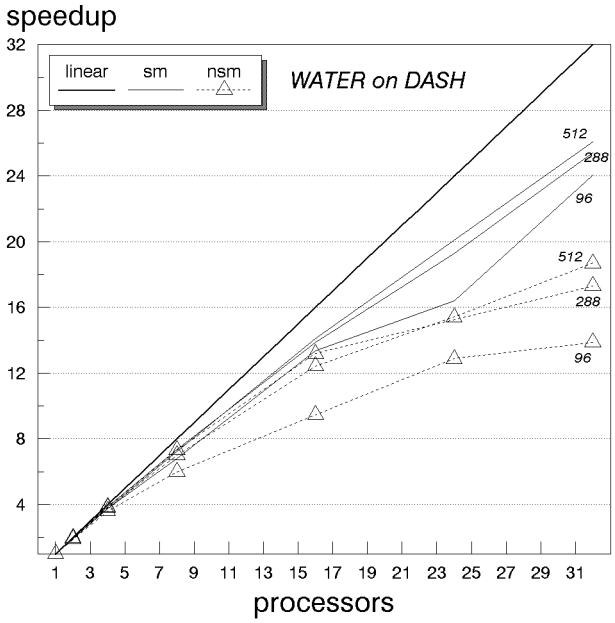


Figure 2.15: WATER speedup: DASH

other hand, the experimental and analytical results suggest that the nonshared memory model is more portable and scalable. At this point, one can envision two possible approaches to meet all three requirements.

1. Choose the shared memory model and develop hardware or compiler optimization techniques to achieve consistent scalability in the user program[Qin & Baer 97].

2. Choose the nonshared memory model and develop new language abstractions that are convenient to program. In addition, the compiler must be able to compile these abstractions efficiently so that the advantage of an accurate model is not affected.

The next chapter will examine the latter approach.

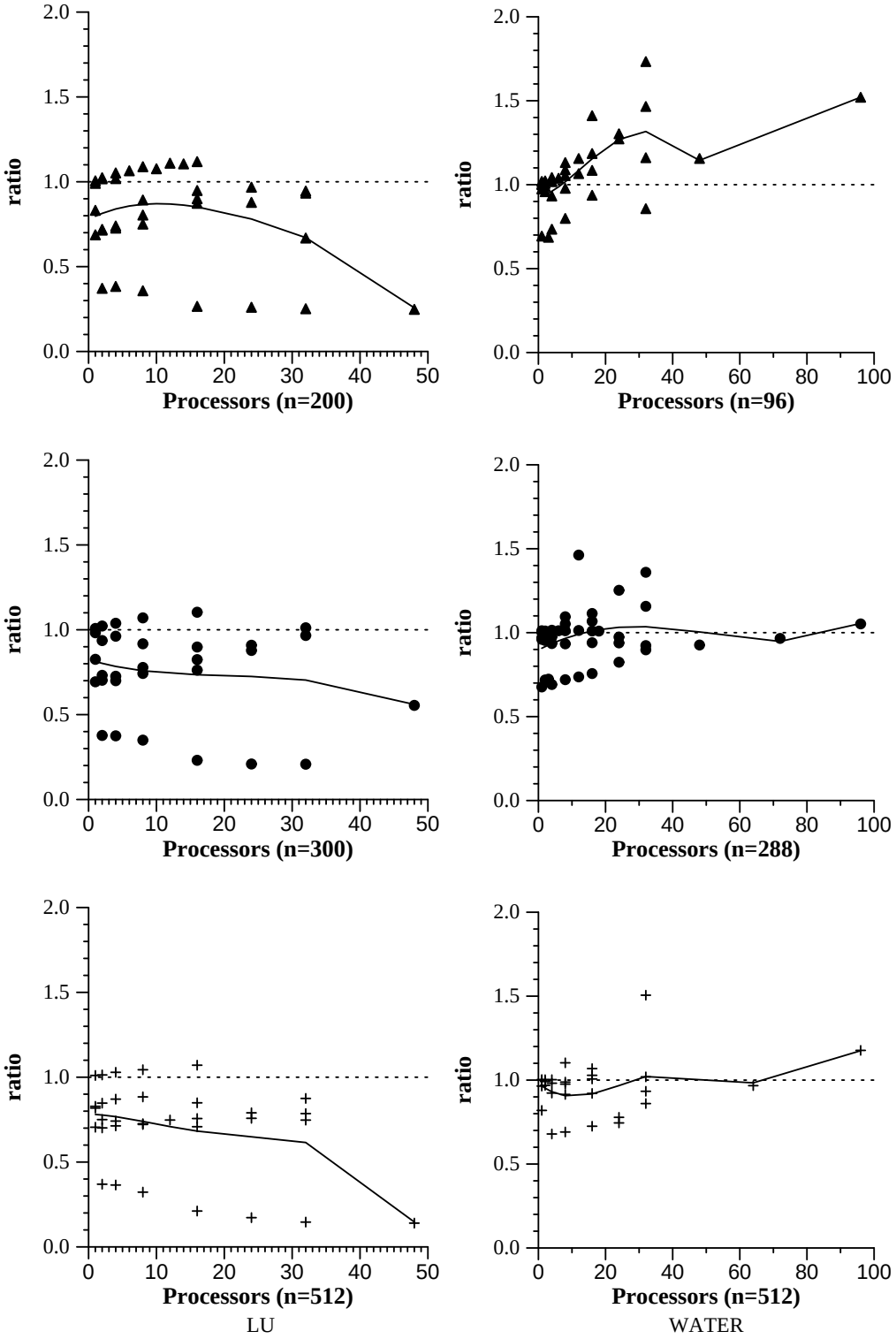


Figure 2.16: $\frac{P_{ns}}{P_s}$ factor for all machines

The solid line is a curve fit of the data points; below the dotted line P_{ns} is better.

Chapter 3

Two Data Parallel Languages, HPF and ZPL

3.1 Introduction

Chapter 2 has shown evidence that a programming language based on the nonshared memory programming model achieves scalability and portability. A language realizes an instance of a programming model by providing a number of abstractions; therefore, the remaining task for the language designer is to achieve ease of use in the design of the language abstraction. A number of current languages and language extensions implement the nonshared memory programming model and in this chapter we will consider two such languages, HPF and ZPL. Although HPF and ZPL share many similarities, their selection in this study is based primarily on their differences. They represent two classes of parallel languages built from fundamentally different philosophies: HPF is an extension to a sequential language while ZPL is designed from first principles.

It is helpful to begin by recognizing that HPF and ZPL are both *universal* in the sense that any computation can be expressed in either language; therefore the issue is not whether a certain computation can be expressed in a language but whether it can be done conveniently. In this respect, a language may be described as being *expressive*,

which generally means that given an algorithm, a programmer can quickly arrive at several possible implementations that are conceptually efficient and that require only a reasonable coding effort from the programmer.

Expressiveness can be achieved in several ways. Designing a language abstraction as a first class object contributes to expressiveness since the abstraction can then be combined and used in many flexible combinations. Abstractions that naturally capture frequently used operations also help to make the language more expressive. Unfortunately, quantifying the expressiveness of a language is problematic because it is a subjective quality that depends on the programmer's style and experience. A programmer who is well versed in a particular language and who is partial to the language is likely to find the language expressive. Some metrics can help to measure expressiveness, for instance the number of lines of code to implement an algorithm, the programmer's productivity, the number of bugs, etc.

In this thesis, we take the modeling viewpoint to adopt the following argument: the convenience factor in a language is only meaningful if the language abstractions can be implemented efficiently and consistently. In other words, this issue must be considered in conjunction with, not in isolation from, the scalability and portability issues (recall Figure 1.1). From this argument, the ease of use can be evaluated by characterizing the performance of the basic abstractions of the language. Clearly, consistent high performance is the ideal case. Even in the case of low performance, if the abstractions are identifiable and behave consistently, a user can attempt to learn the behavior and supplement the programming model. However, if the performance is unpredictable, then one must conclude that the language is not easy to use.

This chapter begins with a general review of HPF and ZPL in Sections 3.2 and 3.3. The reader who is unfamiliar with either language is referred to the appropriate language reference and user guide for further detail [Snyder 94, Forum 93]. A discussion follows in Section 3.4 and 3.5 that compares in detail the language features that HPF and ZPL provide for parallel programming. Since we are primarily interested in the parallel case,

we will assume that the non-parallel aspects are equally adequate in both languages. Table 3.1 summarizes the main points to guide the discussion, namely how parallelism is expressed (computation and array reference) and distributed (data and computation).

Table 3.1: High level comparison of HPF and ZPL

Expressing	ZPL	HPF
Parallel computation	region	do loop + directives
	array operation	forall, where
		F90 array
		F90 intrinsic
Sequential computation	for	do loop
Local array	index	index
Nonlocal array	direction, at	index
Distributing	**ZPL**	**HPF**
Data	block	(block,cyclic,blockcyclic)
subroutine boundary	no communication	redistribute
Computation	owner-computes	implementation dependent

Figure 1.3(b) depicts portability as multiple compiler implementations for a language. Then to evaluate the portability of the language requires multiple compilers on multiple machines since the goal is to establish whether consistent performance can be achieved on different platforms. HPF satisfies this requirement since multiple compilers exist on several machines. For ZPL, the existence of only one compiler implementation appears to limit an evaluation on portability, even though the compiler supports many platforms. However, portability is strongly coupled with scalability and ease of use. In this respect, ZPL is based on a clear performance model that requires any compiler implementation to adhere to (Chapter 4); therefore, portability is an inherent quality of ZPL.

3.2 A review of HPF

In the last decade, trends in VLSI technology have been driving parallel machines toward physically distributed memory with nonuniform access time. As a result, data locality becomes critical to high performance in this class of machines. In optimizing for locality, automatic data partitioning proved to be a very difficult problem. As a result, a number of computer vendors and university projects began introducing various forms of user-specified directives to aid the compilers in partitioning the data. For instance, MASPAR Fortran provides *CMPF MAP*[Joi 95]; CM Fortran uses *CMF$ LAYOUT*; Cray's CRAFT Fortran uses *CDIR$* [Thi 94]. Others include Fortran D[Hiranandani et al. 94], and Vienna Fortran[Benkner et al. 92].

Recognizing this trend, the High Performance Fortran Forum was formed to develop a standard that was finalized in 1994 as the High Performance Fortran Language specification [Forum 93]. HPF quickly became a far reaching effort involving a wide consortium of companies and universities[Forum 93]. In this respect, one major contribution of HPF is that it is the first recognized standard in parallel language. HPF's strategy is to extend from the Fortran 77 and Fortran 90 base with directives and several new language constructs. The directives specify the data distribution as well as other aspects of the base Fortran language that are affected by parallelization. To maintain compatibility with the standard Fortran, the directives are specified as comments.

The attraction of HPF is manifold. First, using Fortran as the base language promises quick user acceptance since the language is well established in the targeted community. Second, the use of directives to parallelize the program implies ease of programming since conceptually the directives can be added incrementally without affecting the correctness of the program. In fact, it is conceivable that a compiler can parallelize the program without any help from the user although this appears to contradict the need for the directives in the first place. On the other hand, there are potential disadvantages. First, a parallel language that is an extension of a sequential language will likely inherit language features that are either incompatible with parallelization or difficult for a compiler to

analyze[Snyder 86]. Second, the optional nature of the directives, while convenient, may present an unpredictable programming model to the users since it may not be clear to what degree the program needs to be annotated with the directives. A program that scales well with a particular HPF compiler may not scale with a different compiler. Thus the latter may present a challenge to the users while the former presents a challenge as well as an opportunity for the compiler developers. Yet, the opportunity to develop new optimization techniques is attractive to both academic researchers who can find direct application of their work and to software vendors who can differentiate their products by their optimization capability.

An HPF programmer would proceed normally with the Fortran programming, then insert the directives as hints for the compiler to parallelize the program. Parallelism in an HPF program comes from the DO loop, the Fortran 90 array operations, and the *WHERE* and *Forall* constructs.

The directives for data distribution support a two-phase process in which an array is aligned relative to a template or another array that has already been distributed, then the template is distributed over a processor grid. An array distribution can be changed at any point by *REDISTRIBUTE* and *REALIGN*. The data distribution can also change implicitly across subroutine boundary since the caller and the callee may specify different distributions for the array. In this case, the *transcriptive, descriptive, prescriptive* directives provide a number of options for redistributing the subroutine arguments.

Some directives provide hints for the data dependence analysis. For instance, *PURE* asserts that the subroutine has no side effects so that its presence in a loop does not inhibit the loop parallelism, and *INDEPENDENT* asserts that the loop has no loop carried dependence, allowing the compiler to parallelize the loop without any further analysis. Other directives resolve conflicts between the original Fortran sequential semantics and parallelization; for instance *SEQUENCE* dictates that the array has to be stored in contiguous memory locations according to the standard Fortran model.

The HPF language committee recognized that compilers supporting the full language specification may require a significant development time; therefore a subset of the language was defined to facilitate earlier compiler implementation.

Current HPF compiler vendors include APR (Applied Parallel Research), PGI (Portland Group Inc.), IBM, and DEC, and the supported platforms include the IBM SP2, Cray T3D, Intel Paragon, DEC SMP AlphaServer, and SGI PowerChallenge. The vendors, however, do not support the same set of HPF features. The PGI and DEC compiler support the full HPF specifications with some exceptions; the APR compiler supports the HPF subset; the IBM compiler supports the subset and several extensions from the full specification. All compilers employ extensive optimizations for communication and parallelization, although a vendor may choose to focus the optimization on certain aspects of HPF. HPF prototypes have also been built in numerous university projects.

The HPF Forum is currently working on the HPF 2.0 specification to correct some shortcomings of HPF 1.1, to standardize some common practices among the HPF compilers, and to broaden the applicability of HPF [Forum 96]. Some notable features include:

- Data distribution: *shadow region* allocates a region surrounding the local partition that overlaps the adjacent partition; *gen_block* allows irregular block distribution; *indirect* allows per-element mapping of array elements to processors.

- Computation distribution: *on home* specifies the owner-computes rule; *reduction* asserts that a loop is a reduction.

3.3 A review of ZPL

Recognizing the limitation of the PRAM as a model for writing parallel programs, Snyder proposed the Candidate Type Architecture (CTA) [Snyder 86]. The CTA model attempts to capture the essential qualities of parallel machines at a level that is neither too low so that it is limited to a small class of machines nor too high so that important

characteristics are lost. The proposed CTA simply consists of a number of processors connected by a sparse network and controlled by a controller; therefore it matches most if not all current parallel architectures.

To program a CTA machine, the concept of Phase Abstractions and the associated language XYZ were developed [Alverson et al. 93]. The language encapsulates the data distribution in *Data Ensembles* and the communication in *Port Ensembles*. The three levels X, Y and Z make up a structured and hierarchical approach to programming: level Z is concerned with solving the problem with algorithms, level Y configures the data and port ensembles to support each algorithm, and level X contains the local computation to implement the algorithm. To make the implementation feasible within the resources of academia, the array language ZPL was proposed in 1993 as a subset of the language XYZ [Lin & Snyder 93], and after several years of development and refinement, a compiler was completed in 1995 [Lin 94, Chamberlain et al. 95].

The distinction of ZPL is that the language is designed from first principles: the freedom from inheriting a sequential language allows new concepts and language constructs to be invented to create a concrete delineation between parallel and sequential execution. Consequently, the programming model presented to the user is clear and the compiler does not have to manage complex interactions between language features or artificial dependencies. One may expect that it is both easier to develop a ZPL compiler and to write a ZPL program that scales well. In return, the tradeoff in designing a new language without any legacy is the challenge of gaining user acceptance.

ZPL introduces several important new abstractions for expressing parallelism, the most fundamental of which is a concept called *region*. A region is simply a rectilinear set of indices. A region is used both to declare the shape of a *parallel array* as well as to specify the index set for a block of statements to operate over (the parallel array is also called an *ensemble*). As an example, consider the following example in which an 8×8 region is declared. Then R is used both to declare an ensemble A and to initialize A to 1:

```
        region    R = [1..8,1..8];
        var       A: [R] integer;
  [R]   A := 1;
```

Each program statement must fall within the scope of a region. The use of a region thus disallows random indexing of the parallel array. This is an important departure from sequential languages since the lack of indexing is a powerful reminder to the programmer that no order exists during the parallel execution. Without indexing, references to other elements in the parallel array are made through *direction*'s, which are literally directional vectors, and the @ operator. Additional operators such as *"of"* and *"in"* allow a direction to be combined with a region to define a new region relative the current region. The index range of a region can be a constant (*static*) or a variable (*dynamic*); the index set can be continuous or can have a stride factor.

Conceptually, parallelism is achieved by overlaying the ensembles over a processor grid, the rank and size of which are set at runtime through command line options [1]. If regions of different sizes exist in the program, a bounding box is computed for all regions and the box is then overlaid over the processor grid.

In addition to the parallel array, ZPL also supports *indexed arrays* which are referenced through explicit indexing. Because a sequential array is replicated, an operation on a sequential array is repeated on all processors, yielding no parallelism. A parallel array on the other hand is distributed; therefore an operation on the array is executed in parallel with each processor performing the operation on the array section it owns (owner-computes).

Being an array language, ZPL operates on the array as one entity. Scalars are promoted as needed to conform to the target array. Scalar procedures and functions are similarly promoted to operate on single array elements but return a full array. Many array operations are supported directly in the language, including *parallel prefix, reduction, flood.* Since these intrinsic functions operate on the entire parallel array, they are

[1] The current implementation supports a 1-D or 2-D processor grid.

optimized to be highly parallel and efficient.

The remainder of the language is largely conventional. ZPL supports the standard data types (*integer, real, double,...*), arithmetic and logical operators, and control construct (*if, for, while,...*). *"config var"*'s are program constant but can be set at runtime.

A compiler has been implemented at the University of Washington and currently supports the Intel Paragon, Cray T3D, IBM SP2, KSR2 and network of DEC Alpha workstations. The compiler implements advanced optimizations such as loop fusion, array contraction, redundant message elimination. The runtime system supports MPI, PVM, and shared memory, employing a novel method called *Ironman* to exploit the best advantages of the particular communication interface.

3.4 Expressing parallelism

3.4.1 Parallel computation

Parallelism in HPF can be derived from the existing constructs in F77 and F90 as well as new language constructs and directives. Users accustomed to F77 can continue to use the conventional DO loop with the expectation that parallelism will be extracted by the compiler. This offers several advantages: (1) the language is directly usable since the user does not need to learn any new language features, and (2) existing F77 programs can be parallelized with little or no change. Parallelizing general DO loops however is difficult because the loop semantics may enforce considerably more dependency than the computation requires[2]. Research in parallelizing compilers dates back to the late 80's, yet has yielded only limited success[Eigenmann et al. 91, Kuck et al. 93]; therefore this raises questions about the feasibility of the approach.

Programs written in F90 can express parallelism through (1) the array semantics, (2) the WHERE construct, and (3) the intrinsic functions. Consider the array statement:

$$A(1:100) = B(200:400:2) + A(50:150)$$

[2]To guarantee the correctness of a computation, an implementation can impose a stricter but never a weaker ordering.

The semantics call for the RHS to be computed completely before the result is assigned to the LHS. The implication is that there is no dependence between the assignment in the LHS and any references in the RHS. The WHERE construct essentially extends the array semantics to include a conditional statement: each element of an array is tested independently and some statements are executed depending on the TRUE or FALSE value of the test. Conceptually, because the array semantics call for the computation of each array element to be performed independently, the parallelism is clear. The intrinsic functions are functions that operate on an array such as SUM(), SPREAD(). While they are not necessarily parallel in concept, the implementations by the compilers or in the libaries can be parallelized and optimized.

The Forall construct is a new feature provided by HPF which has a semantics analogous to arrays, but allows much more flexible indexing of the array. However, the Forall has restrictions and must be used with care since the flexible indexing may be in conflict with the array semantics and yield undefined results. For instance the syntax allows several values to be assigned to the same array element, but the result will be undefined.

Finally, the $INDEPENDENT directive asserts that the iterations in the immediately following DO or Forall loop can be performed independently, freeing the compiler from any further dependence analysis.

The many ways to express parallelism help make HPF highly expressive in the sense that given a problem, a programmer can quickly implement a solution in one or several possible ways. This expressiveness arises from HPF's compatibility with F77 and F90, as well as new language features. However, implementing each of these features requires a significant effort. This wide range of choices may force a compiler, out of practical considerations, to focus on a particular expression of parallelism. The compromise will in turn lead to nonportable differences between platforms. For instance, the APR compiler targets the market of existing F77 codes by investing its effort in analysis and optimizations for DO loops and performance tuning tools to help the user in restructuring the program. The PGI compiler on the other hand targets new codes written in F90[Bozkus

et al. 95], therefore its ability to analyze DO loops is quite limited.

One method to manage the different HPF constructs is to convert them to a common form, but this can lead to a suboptimal solution. For instance, a compiler may scalarize the array statements in an F90 program and convert it to the same internal representation as an F77 program so that the same analysis applies; however, doing so may introduce artificial dependencies that render the analysis more difficult. Another example is idiom recognition[Gupta et al. 95], where the compiler scans the internal representation to detect patterns of DO loops that implement common global array operations (e.g., summation, reduction). Once detected, they can be substituted with the standard intrinsic functions.

Compared to HPF, ZPL achieves parallelism by employing the *region* concept singularly and pervasively throughout the language. A region is by definition an unordered index set. A region is used to define the scope for both data and computation; therefore computation over a region proceeds in an unordered manner. In addition, built-in operators such as parallel prefix and reduction operate over a region and have a parallel implementation although they may be conceptually sequential. The region and the array operators in ZPL are somewhat analogous to the F90 array statements, the *Forall* construct and the intrinsic functions in HPF, except ZPL's region is applied uniformly throughout the language.

The single mode for expressing parallelism in ZPL does require a user to reason in a new paradigm that is different from the conventional sequential programming. In this paradigm, the user thinks in terms of "do this to all elements at once"; in fact with this array perception, ZPL programming has been likened to programming in Matlab[3]. It may appear that having a single mode for expressing parallelism limits the expressiveness of the language. However, the popularity of Matlab has demonstrated that programming by array is a convenient and expressive tool. It follows that a Matlab user would find ZPL an expressive language, or conversely, once a user has become familiar with array

[3]Matlab however does not offer the notion of distributed computing

programming, implementing a solution in ZPL would be quick and convenient.

Finally, it has been mentioned that unlike HPF, the availability of only one ZPL compiler may limit an evaluation of the portability of ZPL. In this respect, the single mode for expressing parallelism helps minimize any possible differences between multiple implementations of ZPL. This does not mean that all ZPL compilers will offer similar performance, rather it ensures that a ZPL program that is parallelized by one compiler will be parallelized by any compiler.

3.4.2 Array reference

The message passing programming model is inconvenient for at least two reasons: (1) nonlocal data requires communication to be brought to the processor that performs the computation, and (2) the local section of the distributed array requires local indices, which in turn requires frequent conversions from the global indices. Data parallel languages solve these two problems by allowing the user to program with a *global view*, in which the distributed array is accessed using global indices and the communication is hidden or encapsulated in some abstractions. In this approach, the task for the compiler is to compute the local indices and generate the necessary communication.

Note that a global view is not synonymous with shared memory: we define the latter to be a subset of the former. Global view refers to the ability to manage a distributed array in the same manner as an array on a uniprocessor. Shared memory allows random access to array elements with no distinction whether they are local or nonlocal. Thus shared memory implements a global view, but a global view can be implemented by other abstractions beside shared memory. This distinction is important because the communication is a major component of the overall performance.

Both HPF and ZPL offer to the user a global view of the data and both generate the necessary communication to fetch the nonlocal data to maintain this global view. HPF and ZPL differ however in whether the communication is *visible* to the programmer.

ZPL has the following unique characteristics. Consider the two ZPL assignment

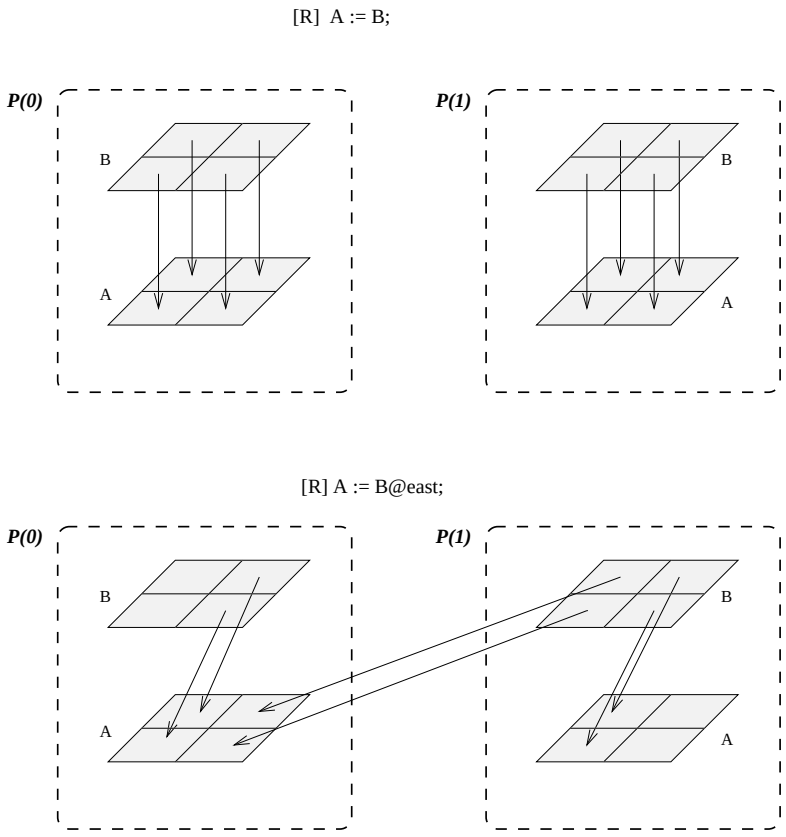


Figure 3.1: Reference to a distributed array in ZPL.
Note: (a) only local data is used in the assignment, and (b) nonlocal data requires communication.

statements in Figure 3.1. By the language semantics, the first statement is guaranteed to involve no communication since all data are local. The second statement requires communication (unless the region is not distributed) because the @ operator accesses data that may not be local.

ZPL semantics dictate this behavior by (1) requiring that all interacting regions be aligned by their indices, e.g., A[1,3] must be aligned with B[1,3], and by (2) replacing absolute indexing with relative indexing, i.e., the @ operator. The first restriction insures that all elements with the same indices will reside on the same processor so that an element wise operation will only involve local data. Communication occurs when the

source and destination indices differ, and the second restriction forces the user to use the special operator when the indices differ. The @ operator thus becomes a visual cue that aids the user in programming an algorithm.

HPF on the other hand provides no visual cue in the language for the occurrence of communication. If the compiler is known to implement the owner-computes rule, it is possible to carefully align the data layout so that the required communication is more visible. However, this requires a conscious effort by the programmer and in the general case, the manual bookkeeping can be quite difficult. For compilers such as APR that can choose to distribute data and computation independently, it is virtually impossible to deduce the communication pattern from the source program.

In a parallel machine, the communication is a major component of the program performance. Given an algorithm, if the compiler can guarantee the optimal communication for this algorithm regardless of how the algorithm is expressed in the source program, then it is an advantage to abstract away the communication from the language and reduce the need to make the communication visible to the programmer. However if such a case is not likely, then the user can only optimize the program if the communication is visible. This criterion is part of the *performance model* which will be explored in more detail in Chapter 4.

3.5 Distributing parallelism

3.5.1 Distributing the data

Data distribution involves the mapping of a section of an array onto a processor in a processor grid. The existence of some distribution methods in a language is clear evidence of the nonshared programming model.

Type of distribution

HPF allows extensive flexibility in distributing the array through the *DISTRIBUTE*, *ALIGN* and *TEMPLATE* directives. The mapping is a two-phase process. First, *ALIGN*

aligns an array relative to another array or a template using an affine function that maps each index from any dimension of the source array to an index in any dimension of the target array. A *TEMPLATE* is a dummy index set that is used solely for computing the distribution. Second, *DISTRIBUTE* specifies how the template is to be distributed over the processor grid. Alternatively, an array can be distributed onto the processor grid directly without the alignment step.

If there are more array dimensions than processor grid dimensions, the remaining array dimensions will be collapsed; if there are fewer array dimensions than processor grid dimensions, the array can be either replicated over the remaining processor grid dimensions or anchored to a specific processor grid dimension. The binding for the data distribution thus occurs at the source level, although at the runtime it is possible to query for the number of processors and make some limited adjustment to the configuration. As an example, Figure 3.2 shows array A(20,20) being distributed by block onto a 2x2 processor grid, and array B(4,4) being aligned relative to array A. Note that B(4,4) maps to a transposed grid because of the swapped index, but within each processor the array section is stored in its normal column major order.

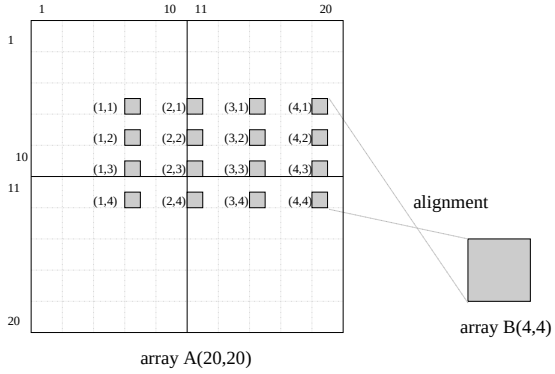
Three types of data distribution are available in HPF: block, cyclic and block cyclic. Block distribution is the most commonly used and the most straightforward to implement. Cyclic distribution is not difficult but can result in significant overhead due to complex index expressions generated by the compiler, particularly when cyclic is mixed with other modes of distribution. Block cyclic distribution is more difficult to analyze and can lead to high overhead as well. This fact is evident in the level of support from the current compilers: the APR and IBM compilers do not support block cyclic.

ZPL's data distribution is more restrictive: only block distribution is supported. Conceptually, all arrays are aligned to one single global index space which is then mapped onto the processor grid. In practice, this global index space is computed as the bounding box for all regions in the program that interact[4]. Distribution is done by pairing the

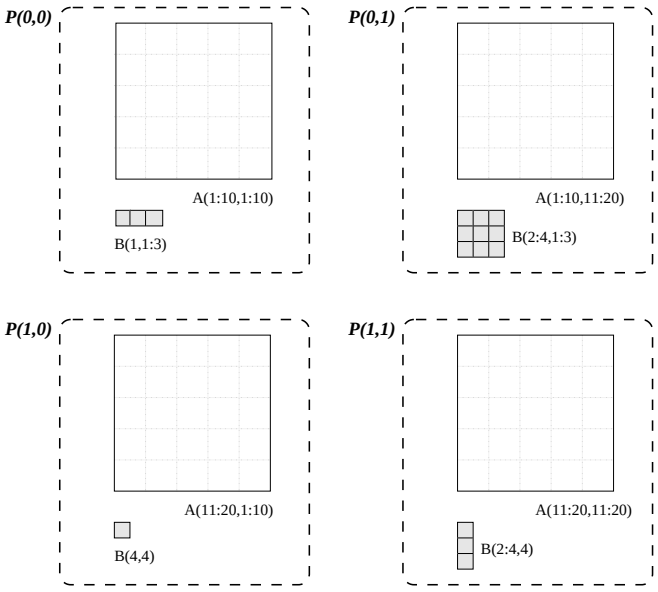
[4]Two regions interact if there exists an assignment statement in which one region is on the RHS and the other region is on the LHS

```
integer A(20,20), B(4,4)
!HPF$ PROCESSOR P(2,2)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO P
!HPF$ ALIGN B(j,i) WITH A(2*i+3, 5*j+23)
```

(a) HPF program segment



(b) Logical mapping



(c) physical mapping

Figure 3.2: Example of HPF data layout.

Note: (1) A(20,20) is distributed by block and B(4,4) is aligned to A; (2) the index is transposed in the mapping function but the array is stored in the normal column major order.

array and processor grid dimensions starting from the rightmost dimension. Given an array rank Ar and a processor rank Pr, if $Ar > Pr$, the remaining array dimensions are collapsed; if $Ar < Pr$, the array is distributed over Ar dimensions on the processor grid and is anchored on the first order of the remaining processor dimensions. The current compiler implementation supports a 2-dimensional processor grid.

An array can be aligned with another by selecting the proper index range and stride of its region. For instance, Figure 3.3 shows array B[9..15,5..11] with a stride of 2 being overlaid over array A[1..20,1..20]. It is possible to reproduce the HPF layout in Figure 3.2, but in general HPF distribution is richer.

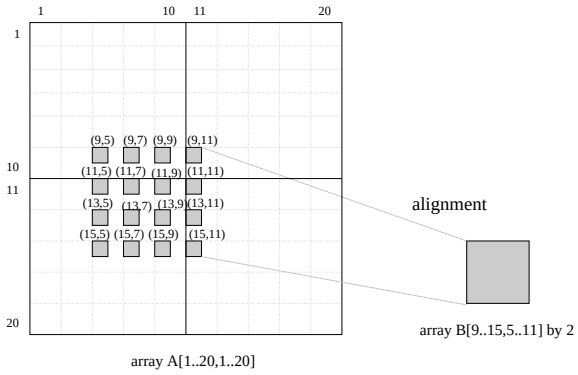
Compared to HPF, ZPL does not map an arbitrary array dimension to a processor grid dimension. This precludes mapping the array to a transposed processor grid as in Figure 3.2. ZPL also does not separate the offset from the index range when two arrays must be laid out relative to each other. For instance, array B in the example has 4×4 elements, but it cannot be indexed as [1..4,1..4]. On the other hand, the processor grid is specified at runtime, delaying the binding and thus allowing different processor configurations to be invoked without recompiling the program.

Redistribution

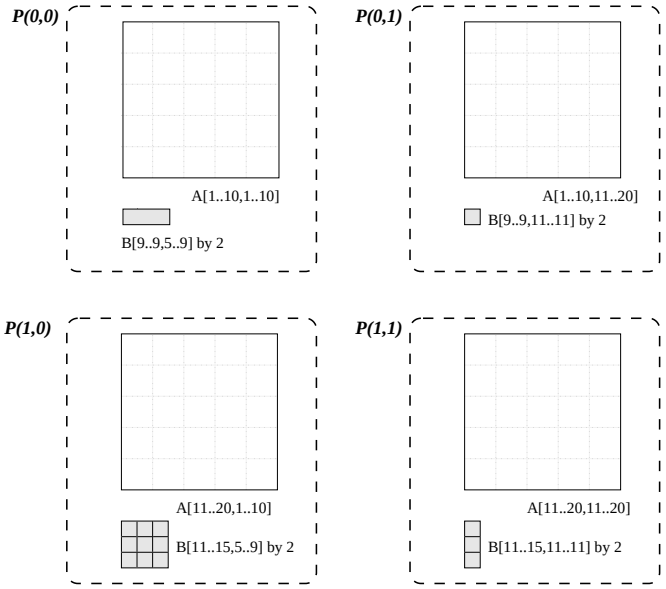
HPF allows an array distribution to be changed in two ways. It can be changed at any point in the program via *REDISTRIBUTE* and *REALIGN* to be tailored to each phase of the computation. In addition, since a subroutine can specify a different data distribution within its scope, an array that is passed as a formal argument may have to be redistributed on the subroutine entry to match the local distribution and then again on the exit to return to the caller's distribution. This presents a potential inefficiency if an array is repeatedly redistributed across the subroutine boundary. This can be avoided if the caller and the callee reside in the same program by ensuring that both specify the same distribution. However for separately compiled subroutines, the caller's distribution is unknown. In addition, the distribution inherited from the caller may be a poor match for the subroutine and result in excessive communication.

```
region bigR[1..20,1..20]
         smallR[9..15,5..11] by [2,2];
var A:  [bigR] integer;
    B: [smallR] integer;
```

(a) ZPL code segment



(b) Logical mapping



(c) Physical mapping

Figure 3.3: Example of ZPL data layout.

Note: the bounding box for A and B is used to compute the partitioning.

HPF does not solve this problem but rather provides the user the options to specify the appropriate action. The normal *DISTRIBUTE* directive is called prescriptive since the array must be redistributed as specified. The *INHERIT* directive is transcriptive: the array distribution is to be inherited from the caller without any redistribution. The *DISTRIBUTE ** directive is descriptive: the programmer asserts that redistribution can be omitted because the distribution in the caller and in the subroutine will be identical, thereby relieving the compiler and the runtime system of the task of checking the distribution. Thus with careful use of these directives, the user can balance between the redistribution cost and the computation in the subroutine.

Procedures in ZPL are distinguished as *sequential* or *parallel*. A procedure is sequential if it does not contain any reference to a parallel variable (a region); otherwise it is a parallel procedure. An example of a sequential procedure is one that accepts an integer and returns its square. This distinction requires the two types of procedure to be treated differently; specifically, the concept of promoting a procedure only applies to a sequential procedure that has no side effects.

Calling a procedure does not involve communication in ZPL for the following reason. Calling a sequential procedure requires no communication since by definition, the procedure only operates on data local to the processor. Calling a parallel procedure without passing any parallel array also results in no communication since parallel arrays declared within the procedure scope are not visible to the caller. If a parallel array is passed, the compiler will analyze references to the parallel array in the interprocedural analysis step to compute the appropriate bounding box for partitioning. The result is that the parallel array will remain perfectly aligned across the procedure boundary, thus requiring no communication.

Discussion

HPF clearly provides a wide range of complex data distributions. In addition, the actual distribution of an array is completely decoupled from the reference to array elements since HPF must adhere to the Fortran syntax; in other words, an array element

is accessed in the same manner regardless of how it is distributed. These features culminate in a high level of convenience and expressiveness for HPF programs. In terms of the modeling framework, they capture the nonshared memory characteristics of the programming model by requiring the user to consciously distribute the data. At the same time they allow the problem to be solved from a global point of view, sparing the user the tedious low level programming typical of message passing programs. Since HPF's abstractions to implement the nonshared memory programming model appear to be reasonable, we proceed to consider whether they can be implemented efficiently and consistently. The challenge to the compiler lies in two areas: optimizing the data redistribution and the nonlocal array references. The latter has been discussed in detail in Section 3.4.2.

Redistribution is clearly a heavy weight process due to the complex mapping and the communication involved. The compiler can generate some code for redistribution, but the runtime system must handle most cases because many variables are unknown. Redistribution requires computing the distribution map of the source and destination arrays, intersecting the maps to determine all the senders and receivers, generate the communication and marshalling the data to and from the communication buffers. The cost thus includes the computation, the buffer allocation and copying, the communication and the degradation in cache performance. For this reason, the APR compiler provides the option of allocating the full storage for a distributed array at each node, of which only the local array section is used. This reduces the overhead for buffer allocation and the data marshalling, but the memory requirement is not scalable.

Recalling our modeling framework, the significant redistribution cost should be visible in the programming model. The explicit *REDISTRIBUTE* and *REALIGN* directives meet this requirement, however the implicit redistribution that occurs across subroutine boundaries does not. Since the default action by the compiler is to enforce redistribution to ensure correctness, a programmer may be surprised by unexpected redistribution calls inserted by the compiler. The presence of *REDISTRIBUTE* or *REALIGN* in the source

program also adds complexity to the compiler since it interferes with many standard optimizations for the sequential code; for this reason, the IBM HPF compiler currently does not support *REDISTRIBUTE* and *REALIGN*.

3.5.2 Distributing the computation

Section 3.4.1 describes how the user can express the parallel computation. The next step is to distribute the parallel computation among the processors and ultimately, the degree of parallelism achieved depends on how this distribution is actually implemented. In the ideal case, the computation will somehow be evenly distributed among the processors with no overhead. In reality however, the distribution strategy not only affects the workload balance but also the necessary communication since nonlocal data needed for the computation must be fetched to the site that performs the computation and the result must be stored to the appropriate site.

Since parallelism in both languages is derived from some form of looping, the common method for distributing the loop is to adjust the loop bounds and stride to reflect the local workload. If the loop cannot be distributed, the bounds will not be adjusted and all processors would execute all iterations, resulting in no parallelism.

Can the loop be distributed?

Before a loop can be distributed, an HPF compiler must first analyze the dependencies to determine whether correctness can be preserved. Clearly the distribution is inhibited if the analysis fails and in this respect, HPF provides several directives to help the compiler. A loop with no cross iteration dependency can be annotated with *INDEPENDENT*. A subroutine call within the loop may have side effects which would constitute a loop dependency; the subroutine with no side effects can be annotated with *PURE* to assert this fact. Whether a user needs to insert these directives depends on the analysis capability of the compiler.

How to distribute?

In contrast to its extensive data layout capability, HPF does not specify how the

computation is to be distributed, leaving this decision to the compiler implementation[5]. One motivation for this choice is to leave open optimization opportunities for the compiler. A common convention adopted by several compilers (IBM, PGI, DEC) is the owner-computes rule: the processor that owns the LHS of an assignment statement is to execute the statement. In this scheme, the compiler will generate the communication necessary for the owner of the LHS to gather the data referenced in the RHS before the computation takes place. The advantage of this scheme is twofold: it is simple to implement and the assignment is made locally.

There are situations where the owner-computes rule is suboptimal. For instance, if many array elements referenced in the RHS reside on the same processor, the communication volume can be reduced by allowing this processor to perform the computation and to send only the result to the owner of the LHS. Various proposals have been made to optimize the communication by selecting the best site to perform the computation [Amarasinghe & Lam 93], but this proved to be a difficult problem. Other schemes are also possible, for instance the APR compiler simply selects (automatically using some set of rules or with the user's assistance) a favorable loop to distribute[App 95].

From the modeling point of view, the absence of a definite scheme to distribute the computation leaves a gap in the programming model. This gap is both necessary to preserve the Fortran syntax and intentional to make the programming convenient. Yet it forces a programmer either to (1) rely completely on the compiler to parallelize the program or (2) close the gap by tuning the program to the specific compiler and sacrificing portability in the process. In other words, it may be difficult to balance the workload in a portable manner since the parallelization strategy varies with the platform. A case study of the NAS EP benchmark in Chapter 5 will illustrate this problem.

ZPL distributes its computation according to a number of rigid rules, leaving little opportunity for deviation or in fact optimization. First, the global region representing the whole program will be partitioned by block onto the processor grid, thereby distrib-

[5]HPF 2.0 addresses this issue by adding the *ON HOME* directive that enforces the owner-computes rule[Forum 96]

uting any arrays and statements defined over the region. Second, the computation is aligned with the LHS, thereby ensuring that any ZPL implementation will follow the owner-computes rule. Distribution in ZPL thus strictly implements the semantics of the language: a ZPL compiler only has to compute the local loop bounds for the array on the LHS; it does not need to perform any analysis to arrive at the decision to parallelize the loop.

The difference between HPF and ZPL can be summarized as follows. An HPF compiler begins with an undistributed loop and attempts to distribute it by analyzing the dependencies and/or by relying on user directives; if the analysis fails, the loop is not distributed and all processors execute the full iteration. ZPL on the other hand strictly implements the language semantics; in other words, a statement within a region scope is a parallel statement and is guaranteed to execute in parallel.

In terms of modeling, the HPF model of parallelization is weaker since it includes two variables: whether the loop can be parallelized and how the workload will be distributed. ZPL model is stronger since the language semantics clearly define the behavior of these two variables.

For HPF compilers that use the owner-computes rule, the computation distribution shadows the data distribution. The wide range of choice for data distribution thus leads to more flexibility in implementing an algorithm. For instance, the LU decomposition would benefit from a block cyclic distribution since the workload is more evenly distributed, while an SOR solution would prefer a block decomposition to exploit the neighbor locality. ZPL's choice for distribution is more limited in this respect since only block distribution is supported.

3.6 Conclusion

The issue is not whether one can achieve performance in a particular language but whether this can be done conveniently. Given time and resources, an ardent enthusiast of a language can tune a program in the language to achieve good scalar and parallel

performance, but it may not be convenient to arrive at the result and the result may not be portable. An example of this can be found in the benchmarks used by some HPF vendors: the APR benchmarks for HPF have been carefully tuned to match the APR compiler; as a result they perform well with the APR compiler but not with other HPF compilers. PGI versions of the benchmarks also exhibit this characteristic.

An optimization technique is most successful when it can cover the majority of the cases. When it only covers specific cases, several problems arise: the case must first occur in the program and then the compiler must be able to recognize the case to apply the optimization. The latter problem can be particularly difficult, but both problems reduce the effectiveness of the optimization. Optimizations for array references in HPF seem to suffer from this phenomenon: the ability to make global array references leads to many possible patterns of reference, with no particular prevalent pattern that can be easily recognized.

Given that parallel language is still a developing area, one can safely presume that the designers of neither HPF nor ZPL assert that their current language is the final solution for parallel programming. The discussion in this chapter clearly shows the benefits and liabilities of two fundamentally different parallel languages. By capitalizing on the Fortran base, HPF succeeds in establishing a standard in parallel languages where none existed before and this should be recognized as a very significant achievement. However, building from an inherently sequential language carries many implications that are now becoming more apparent as experience is gained from implementing and using the language. One general observation is that the flow of information in an HPF program is largely from the user to the compiler. In other words, the user provides information about the program to aid the compiler, but the compiler guarantees to the user little more than correctness.

ZPL is based on a model of parallel machines. Being designed from first principles, ZPL can freely incorporate new concepts and language constructs that work well. Because these concepts and constructs are intrinsic to the language, the language speci-

fication serves as a secure contract to the user that the program will behave as expected. Information thus flows both ways, from the language to the user and vice versa. Being new also carries the burden of having to prove ZPL's merit and to gain acceptance from users and software developers.

In the following chapters, we will further compare HPF and ZPL, first with respect to the concept of performance model (Chapter 4), then experimentally using the current HPF and ZPL compilers (Chapter 5).

Chapter 4

The Performance Model in HPF and ZPL

> "HPF must be used carefully, because efficient and inefficient codes look very similar. An HPF translator generates code to implicitly perform whatever communication and overhead required to correctly execute the HPF program statements. On the other hand, with a program parallelized by explicit insertion of message passing communication calls it is very apparent where overhead is introduced. The ability of a translator to generate whatever communication and overhead is required, is in a way a defect, in that it obscures what really happens at the runtime."
> *APR Fortran Parallelization Handbook [Friedman et al. 95]*

4.1 Introduction

In Chapter 1 we discussed the pervasive use of modeling. Having examined qualitatively in Chapter 3 the facilities in HPF and ZPL for parallel programming, we find that a clear distinction between HPF and ZPL is in the modeling aspect of the languages. In this chapter, we begin by identifying this modeling aspect, namely the *performance model*.

One may question whether this difference ultimately leads to any tangible difference in the program performance. In other words, *what is the benefit of a performance model?* We will seek a quantitative answer to this question by first formulating a methodology and then by considering two in-depth case studies.

4.1.1 The performance model

Recall that a model reflects our understanding of how a certain system operates; it captures the information that is necessary and sufficient for us to use the system. It follows that if the model is inaccurate or insufficient, it will be difficult to use the system effectively. Conversely, too much complexity makes the model difficult to use and defeats its very purpose. A model must capture the right level of information. In particular, when performance is an objective, the model must capture some information about performance.

For a concrete example, consider sequential languages such as C or Fortran. Proficient programmers in these languages have an approximate understanding of how each language abstraction correlates with its low level implementation, and they use this information routinely without any special consideration. For instance, procedure calls are convenient and helpful in structuring a large program, however, they do incur cost in pushing the stack and passing arguments, and this cost is befittingly associated with the text required to set up and call a procedure. As a result, a programmer would not create procedures unnecessarily, nor would he/she create procedures thinking that doing so would make the program run faster.

In the context of program development, this approximate understanding helps form a coarse but reliable classification of language abstractions in terms of their cost (execution time). We call this relative ranking the *performance model*. The performance model is coarse because it is not possible to use it to determine the actual execution time of the program independent of the targeted machine. In this respect, it is different from parameterized, analytical models such as the formula that is commonly used to predict

the communication cost:

$$time = startup + \frac{msgsize}{bandwidth}$$

The performance model is reliable because a language abstraction classified as expensive will always take more time than one classified as inexpensive. For this reason, a more suggestive name for the model may be "WYSIWYG". For the terminology, Snyder in previous work [Snyder 95] defines a *machine model* as a set of common facilities for a large class of parallel machines; the set contains sufficient detail to allow the programmer to choose between programming alternatives. A *programming model* then extends the machine model by adding new abstractions with "known" costs. For the focus of this chapter, the term performance model refers to the performance aspect of Snyder's programming model. In other words, given the programming model, the performance model defines the relative cost of the abstractions.

Revisiting C and Fortran from above, their performance model arises naturally from the close affiliation between the language and the hardware. The abstract von Neumann machine, by encompassing most sequential machines, thus emerges as the machine model for these languages.

How is the performance model used?

The performance model indicates the relative performance; therefore it helps the programmer in two important tasks: (1) given several alternative algorithms for a problem, how to choose the appropriate algorithm, and (2) given several alternatives to implement an algorithm, how to choose the appropriate implementation. These two tasks are critical because ultimately, the most effective optimization rests with the user. No compiler optimization can transform a poor algorithm into an optimal algorithm.

How does a language incorporate a performance model?

Because no explicit formula or equation is involved, a performance model is typically derived from several sources. For a frequently used programming technique that has been encapsulated in a high level abstraction, its implementation and therefore its cost is well understood. An example is the DO loop. The language semantics, if well defined

and concise, can provide valuable information. For instance, the *register* directive in C suggests that the attributed variable can be accessed faster than normal variables. Visual cues in the language syntax help to associate a relative cost with a particular language construct. For instance, the quantity of text can serve as the cue: a construct that is fast (e.g., an arithmetic operation) should be expressed with little text while a slower construct (e.g., a library function call) should require more text.

Such a loose association leads to the coarse quality of the model, while the reliability of the model depends on the accurate association between the construct and the actual cost. The only requirement is that the part of the language that needs to have a cost associated must be visually identifiable.

Is the performance model a standard component of all languages?

Many languages exist that do not contain a performance model. For example, Prolog was designed for expressing predicate calculus. Because there is no performance information in the language construct, it is difficult to infer whether a Prolog statement will execute quickly or take an exponential amount of time. Similarly, dataflow and functional languages are built from mathematical abstractions with little consideration for the physical implementation. As a result, no performance model exists to guide the users in writing dataflow or functional programs that can be emulated efficiently on conventional machines.

4.1.2 ZPL's performance model

Through extended research in programming models, ZPL designers recognized the existence and the importance of the performance model as a distinct entity. ZPL was designed with the performance model as a clear objective; as a result, one of ZPL's significant contribution is the careful integration of a performance model into the language.

ZPL's machine model (the CTA) and programming model have been described earlier. For the performance model, ZPL abstractions exhibit the following behavior, listed in the order of highest to lowest performance [Snyder 94].

1. Element-wise array operations execute fully in parallel with no communication and achieve the highest performance. These operations are clearly distinguished since they involve only arrays declared as parallel arrays.

2. @ operation is likely to involve communication: the cost can be approximated as one message per @.

3. Flood operation requires one or more messages to replicate an array section; it is identified by the flood operator >>.

4. Reduction and scan operation require collective communication that involves multiple messages; they are identified by the operators << and ||.

5. Scalar computation is replicated on all processors and achieves no speedup.

6. The permute operator ## and I/O are the most expensive operations; the former typically involves all to all communication, while the latter involves physical devices that are several orders of magnitude slower than the processor.

4.1.3 HPF's performance model

Although Fortran has an effective performance model for sequential machines, the requirement to remain compatible with Fortran limits the parallel performance model to the set of directives for distributing the arrays and controlling some communication and data dependences. Beyond this, the parallel performance depends entirely on the optimization capability of the compiler. The nature of the directives themselves weakens the performance model: because a compiler is not obligated to implement a directive, the user is not guaranteed of its benefit. The major components of HPF's performance model include (not listed in any order):

- $DISTRIBUTE$, $ALIGN$ are used to distribute the arrays.
- $INDEPENDENT$, Forall and F90 array syntax allow the computation to proceed fully in parallel; however the communication cost is not visible.

- Redistribution requires expensive mapping and communication. It can be identified with $REDISTRIBUTE$. Redistribution for subroutines can be controlled with some directives, but is otherwise not visible.

- Scalar computation is replicated on all processors and achieves no speedup.

An issue worth addressing is the effect of compiler differences on the performance model. We must distinguish between the performance model established by the language specification and a performance model that has been supplemented with knowledge about a specific compiler. The former is portable while the latter is not. Clearly, optimization strategies vary and differences will always exist between compilers. However, a concise performance model in the language will maintain a consistent language behavior in the face of compiler differences. Otherwise, the users may be forced to drastically change the programming style to accommodate a compiler.

Consider the Fortran storage model which specifies that for physically consecutive array elements in memory, the leftmost index changes the fastest. This model restricts the compiler from using a different storage layout and may prevent some types of optimization, but it guarantees the users a certain behavior. As a result, a programmer can structure the code to take advantage of the spatial locality in the array references. Furthermore, the programmer can expect that the code optimized in this manner will never perform worse than an unoptimized code, regardless of the compilers.

4.1.4 A methodology to evaluate the performance model

We now formulate a way to quantify the benefit of the performance model. The notation $\underline{r}$ and $\underline{f}$ are used to denote a relation where $\underline{r}, \underline{f} \in \{>, <, \approx\}$.

Consider two alternative programs P_a and P_b written in a language L to solve a problem. A compiler for the language generates an implementation for each program, resulting in an execution time of T_a and T_b for P_a and P_b, respectively, with a relation of:

$$T_a \ \underline{r} \ T_b$$

Similarly, another compiler for the same language yields:

$$T'_a \ \underline{r'} \ T'_b$$

Define the performance model as the component of the language L that enables the programmer to differentiate between P_a and P_b in terms of their performance. In other words, the model is a function:

$$\mathcal{F}(P_a, P_b) = \underline{f}$$

$\mathcal{F}$ is undefined if the performance model does not exist.

Note that at this point, we are still considering $\underline{f}$, $\underline{r}$ and $\underline{r'}$ as independent of each other. Examine the relations $\underline{r}$ and $\underline{r'}$:

1. If $\underline{r}, \underline{r'} \in \{\approx\}$, then the compiler has in effect neutralized any differences between P_a and P_b, regardless of whether the difference arises from the algorithm or the language construct. In this case, the programmer is relieved of the difficult task of choosing between P_a and P_b and the performance model $\mathcal{F}$ is irrelevant. The reverse must also hold: if $\mathcal{F}$ is undefined, then the compilers must ensure:

$$\underline{r}, \underline{r'} \in \{\approx\} \tag{4.1}$$

   since otherwise the programmer has no means to differentiate and choose between P_a and P_b.

2. If $\underline{r}, \underline{r'} \in \{>, <\}$, then the program with the best performance should be chosen. In this case, the selection process requires the performance model $\mathcal{F}$. It follows that the benefit of $\mathcal{F}$ is equivalent to the performance difference $\Delta(T_a, T_b)$. Furthermore, for $\mathcal{F}$ to be meaningful, the language and the compilers must ensure:

$$\underline{f} = \underline{r} = \underline{r'} \tag{4.2}$$

An interesting corollary follows from equation 4.2: if $\underline{r} \neq \underline{r'}$, then a contradiction occurs, therefore $\mathcal{F}$ must be undefined.

The formulation above allows us to devise a simple methodology to evaluate the performance model.

1. For a given problem, we consider several alternative solutions in a language.

2. The performance for each solution is measured using several compilers to determine the set of relations $\{\underline{r}, \underline{r'}, \ldots\}$. The implementations should also be analyzed to confirm the relations.

3. If equation 4.1 is satisfied from $\{\underline{r}, \underline{r'}, \ldots\}$, then $\mathcal{F}$ is irrelevant and no further consideration is necessary.

4. Otherwise, we proceed to verify equation 4.2. The performance model $\mathcal{F}$ provides $\underline{f}$ if it is defined.

5. If equation 4.2 holds, then $\mathcal{F}$ has ensured a performance improvement of $\Delta(T_a, T_b)$. Even if $\mathcal{F}$ is undefined, the fact that the remainder of the equation holds will be valuable since it implies that it may be possible to determine $\mathcal{F}$ experimentally.

6. If equation 4.2 does not hold, then the potential performance loss is $\Delta(T_a, T_b)$

We mentioned earlier that the performance model helps the programmer in two tasks: (1) choosing the best algorithm, and (2) choosing the best implementation. For the remainder of this chapter, we apply the methodology above in two in-depth case studies that represent these two tasks. Section 4.2 addresses the implementation choice and Section 4.3 the algorithm choice. ZPL is not involved in the analysis in Section 4.2, but its performance data is provided as a convenient point of reference. Then we discuss current methods to supplement the HPF performance model in Section 4.4, and Section 4.5 gives the conclusions. It should be noted that while the analysis in this chapter involves three HPF compilers, the intention is not to compare the compilers. Rather, the compilers provide the different platforms to evaluate the portability of the language.

4.2 Selecting an implementation: array assignment

For a case study, the simple assignment statement when applied to a distributed array is a good candidate for several reasons. First, the assignment statement is the most elementary component of any language; in a data parallel language particularly, its behavior can be magnified when operating on an entire array. Second, a principal component in the performance model for a parallel language is the communication. In an assignment statement we can minimize the computation to focus on the effect of the communication.

4.2.1 Quantifying the performance model

Following our methodology, the given problem is the array assignment statement. The index expression used in the assignment can be further broken down into 11 types [Bozkus et al. 94], thus dividing the main problem into 11 sub-problems.

For HPF, the alternatives to express array assignments include the conventional DO loop, the F90 array syntax or the Forall construct (refer to Section 3.4.1). Among the choices, the DO loop specifies the most dependencies, while the F90 array and the Forall loop have the least. For the latter two, Forall is more expressive than F90; therefore a reasonable expectation is that the Forall loop is the best general choice. If compiler analysis were perfect, one could expect that a compiler would determine that there is no true dependence in these assignments. In this case, the compiler would yield the same performance for all three implementations and equation 4.1 would be satisfied automatically.

For ZPL, the semantics allow only one way to express the assignment statement. Table 4.1 lists the types of index expressions along with their program alternatives in HPF and ZPL. A brief description for each type follows.

The case of *no communication* is the simplest case since the RHS and LHS are aligned exactly. The compiler is expected to be able to determine that the statement involves only local data motion and avoid unnecessary communication.

For the *static shift* case, the RHS and LHS are offset by a constant. To move the RHS

to the LHS requires communication between neighboring processors, but fortunately the optimization is straightforward and easily recognized [Choi & Snyder 97]. The user may expect communication for multiple array elements to be combined (message vectorization). Since the shift amount is known at compile time, an overlap area can be preallocated in memory that is contiguous to the local section of the array. When the statement is encountered, data from the neighboring processor can be copied directly into the preallocated area without a runtime buffer allocation. A variation of the *static shift* case has a coefficient with the RHS index, which requires a gather before the communication.

The *dynamic shift* case is similar to *static shift* except that the shift amount must be computed at runtime; therefore it is more difficult to preallocate the overlap area at compile time, but message vectorization is still possible. When the RHS index has a coefficient, the communication will require a gather operation.

Multicast requires selective communication; it involves replicating the data among a set of processors (usually row or column of the processor grid). The compiler can generate code for a tree broadcast scheme or simply take advantage of the communication library if it provides this function.

The *point to point* case is simply random communication for which optimization is less likely.

The *dynamic index* case involves complex data movement in which the index for one array is a complex but computable function of the other array index. In the precomputation read case, the complex index is in the RHS; therefore the index must be computed before the RHS is fetched. For the postcomputation write case, the complex index is in the LHS; consequently, it must be computed to store the result. The DO loop is well suited for expressing the index expression because it is not always possible to derive an equivalent F90 and Forall implementation, although in our experiments we chose a function that allows all three implementations. Message vectorization for this case is more difficult because the array index must be computed from a complex function. There are

several implementation approaches: the processors can perform an all-to-all broadcast, each element can be fetched individually, or each processor can precompute the index set for the full loop and then perform a scatter or gather using the index set.

The *indirect index* case differs from *dynamic index* in that the index set is data dependent. Assuming that the index set is also distributed, the processors will have to first broadcast to obtain the index set, then the actual data can be obtained in a second communication phase, which like the *dynamic index* can be another all-to-all broadcast, individual messages, or a library scatter/gather call.

In the performance measurement, the assignment involves two linear arrays of N elements that are distributed by block onto a 1-D processor grid and that are perfectly aligned with each other. Since the statements contain no computation, the principal cost is in the data movement and the runtime overhead. The task for the compiler is to generate the communication to fetch the RHS from and store the LHS to the respective owners. Note that more complex scenarios are possible such as higher dimensional arrays and processor grids, cyclic and block cyclic distribution, complex alignment between the arrays. However, these demand more difficult analysis and optimization and are not likely to show a more consistent performance model than the simpler configuration.

The set of assignments is measured on the IBM SP2. The compilers used in the experiments include the HPF compilers from APR, IBM and PGI, and the ZPL compilers from the University of Washington. For each compiler the default optimization option is used – in every case it includes the most aggressive option. A barrier synchronization is used at the beginning of each assignment operation to synchronize the processors. The tracing facility UTE is used to collect a trace per processor, which includes communication calls, timestamps and markers for each program section. The traces are then processed to compute the desired statistics.

4.2.2 Results

Performance model

Table 4.1: Array assignment in HPF and ZPL

1. See dynamic index, RHS; can also be implemented with strided region by renaming the index space.
2. Parameters: T=3, S=2, N=100
3. c, d are variables. A,B,V are N-elements arrays. F is a floodable array [*].
4. All arrays are distributed.
5. For ZPL: region R=[1..N]; direction east = [+S]; east1 = [+1];

Index	Do loop	F90	ForAll	ZPL
no comm.	do i=1,N A(i) = B(i) enddo	A(1:N) = B(1:N)	forall (i=1:N) A(i) = B(i)	[R] A := B;
static shift	do i=1,N-S A(i) = B(i+S) enddo	A(1:N-S) = B(S+1:N)	forall (i=1:(N-S)) A(i) = B(i+S)	[R] A := B@east;
static shift + stride	do i=1,(N-S)/T A(i) = B(i*T+S) enddo	A(1:(N-S)/T) = B((T+S):N:T)	forall (i=1:(N-S)/T) A(i) = B(i*T + S)	(1)

dynamic shift	do i=1,N-c
	A(i) = B(i+c)
	enddo
		A(1:N-c) =
		B(1+c:N)
			forall (i=1:N-c)
			A(i) = B(i+c)
				for I:=1 to c do
				[R] A := A@east1;
				end;

dynamic shift + stride	do i=1,(N-c)/d A(i) = B(i*d+c) enddo	A(1:(N-c)/d) = B((d+c):N:d)	forall (i=1:(N-c)/d) A(i) = B(i*d + c)	(1)
multicast	do i=1,N A(i) = B(S) enddo	A(1:N) = spread(B(S), 1,N)	forall (i=1:N) A(i) = B(S)	[F] A := >> [S] B;
point to point	A(1) = B(N)	A(1) = B(N)	A(1) = B(N)	[F] Af := >> [N] B; [1..1] A := Af;

dynamic index, RHS	do i=1, N
	if (i.le.N/2) then
	j = 2*i-1
	else
	j = (i-N/2)*2
	endif
	A(i) = B(j)
	enddo
		A(1:N/2) =
		B(1:N:2)
		A(N/2+1:N) =
		B(2:N:2)
			forall (i=1:N/2)
			A(i) = B(2*i-1)
			forall (i=N/2+1:N)
			A(i) = B((i-N/2)*2
				if (Index1 <= N/2)
				then
				V := 2*Index1-1;
				else
				V := (Index1-N/2)*2;
				end;
				A := <##[V] B;

dynamic index, LHS	do i=1, N
	if (mod(i,2).eq.1)
	then
	j = (i+1)/2
	else
	j = (i+N)/2
	endif
	A(j) = B(i)
	enddo
		cannot
		be expressed
			forall (i=1:N,
			mod(i,2).eq.1)
			A((i+1)/2) = B(i)
			forall (i=1:N,
			mod(i,2).ne.1)
			A((i+N)/2) = B(i)
				if (Index1%2 = 1)
				then
				V := (Index1+1)/2;
				else
				V := (Index1+N)/2;
				end;
				A := >##[V] B;

indirect index, RHS	do i=1, N A(i) = B(V(i)) enddo	A(1:N) = B(V(1:N))	forall (i=1:N) A(i) = B(V(i))	A := <##[V] B;
indirect index, LHS	do i=1, N A(V(i)) = B(i) enddo	A(V(1:N)) = B(1:N)	forall (i=1:N) A(V(i)) = B(i)	A := >##[V] B;

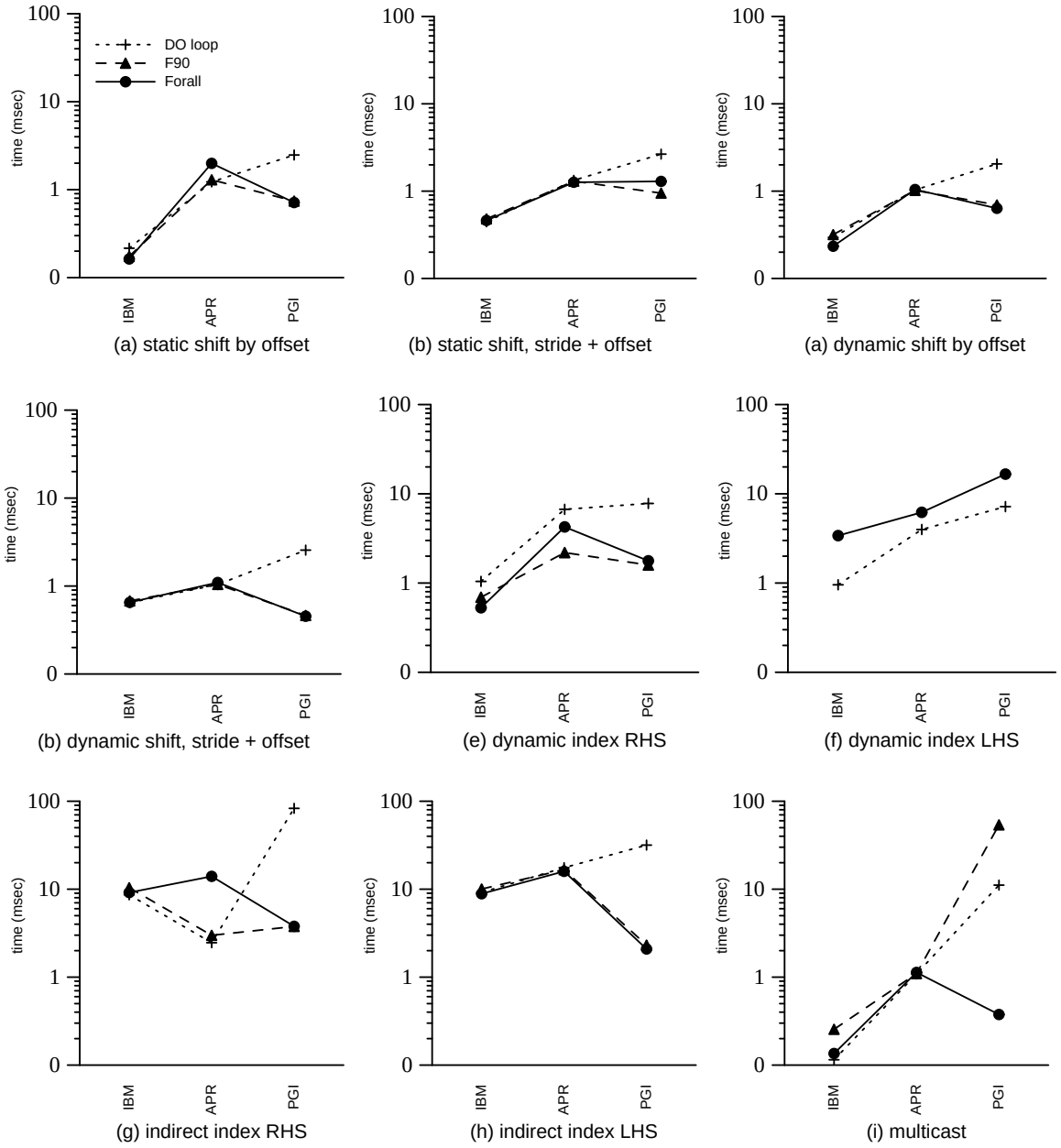


Figure 4.1: Cross compiler performance for HPF array assignment.

Note: 8 processors with different index expressions.

In Figure 4.1, the execution time for the three HPF alternatives (DO loop, F90 array and Forall) is plotted for each subproblem for p=8 (note that the time axis is a log scale due to the wide spread). A quick inspection reveals that the performance data does not support equation 4.1 except for some isolated subsets. In other words, significant performance difference exists between the alternatives.

Proceeding to equation 4.2, we find that the indices in an HPF array reference yield no visual indication of the communication required because they are random and global. The absence of a performance model leaves the relation $\underline{f}$ in equation 4.2 undefined. However, we are still interested in determining whether the remainder of the equation holds since it would be an indication of a possible performance model. The test for $\underline{r} = \underline{r'}$ is shown graphically by connecting the performance for each alternative along the compiler dimension. Equation 4.2 would hold if the performance lines do not intersect, thereby maintaining the same relative order. This would indicate that the best choice is the same for all compilers.

Dynamic index LHS is an example that clearly satisfies equation 4.2 (Figure 4.1(f)): a user would choose the DO implementation since it gives the best performance on all compilers. The Forall construct surprisingly is not the best choice in this case although its semantics specify fewer dependences. Note that this particular index expression does not allow an F90 implementation.

For the other subproblems, the answer is less clear. Table 4.2 tabulated the ordering for each type of index expression. Several other cases also show a consistent performance behavior: two cases clearly favor the F90 implementation, two cases favor the Forall, and four cases show F90 and Forall as equally likely choice. One case favors the DO loop consistently over the Forall despite the dependences. In two cases, the lines intersect and no choice is apparent.

The data does not suggest any single construct as the best choice for all types of array assignment. When considered as individual cases, the F90 and Forall alternatives are better in many cases: clearly the more restrictive semantics in terms of dependences

Table 4.2: Choices of array assignment in HPF.
Note: (1) Intersecting lines indicate no clear choice.
(2) dynamic index LHS cannot be expressed in F90.

Assignment	First	Second	Third
no communication	Forall or F90		DO
multicast	Forall	DO	F90
point to point	Forall or F90		DO
static shift	F90	Forall, DO intersect	
static shift + stride	F90	Forall	DO
dynamic shift	Forall	F90	DO
dynamic shift + stride	Forall or F90		DO
dynamic index,RHS	Forall, F90 intersect		DO
dynamic index,LHS	DO	Forall	
indirect index,RHS	Forall, F90, DO intersect		
indirect index,LHS	Forall or F90		DO

allow the compilers to generate an efficient implementation more easily. However, the richer functionality of Forall leads to a higher overhead than the F90 construct in some cases; as a result, it is not clear when each construct should be used.

Since each construct offers a different level of expressiveness, the situation may arise where it is not possible to express the computation using the construct with the best performance. In the cases where Forall and F90 are equal alternatives, making the second choice is straightforward. For other cases, the second choice is less clear: *Static shift*'s Forall and DO lines intersect because of APR's implementation.

Communication

When an algorithm is analyzed, a coarse but convenient performance metric is the count of messages and collective communication calls since the message startup time

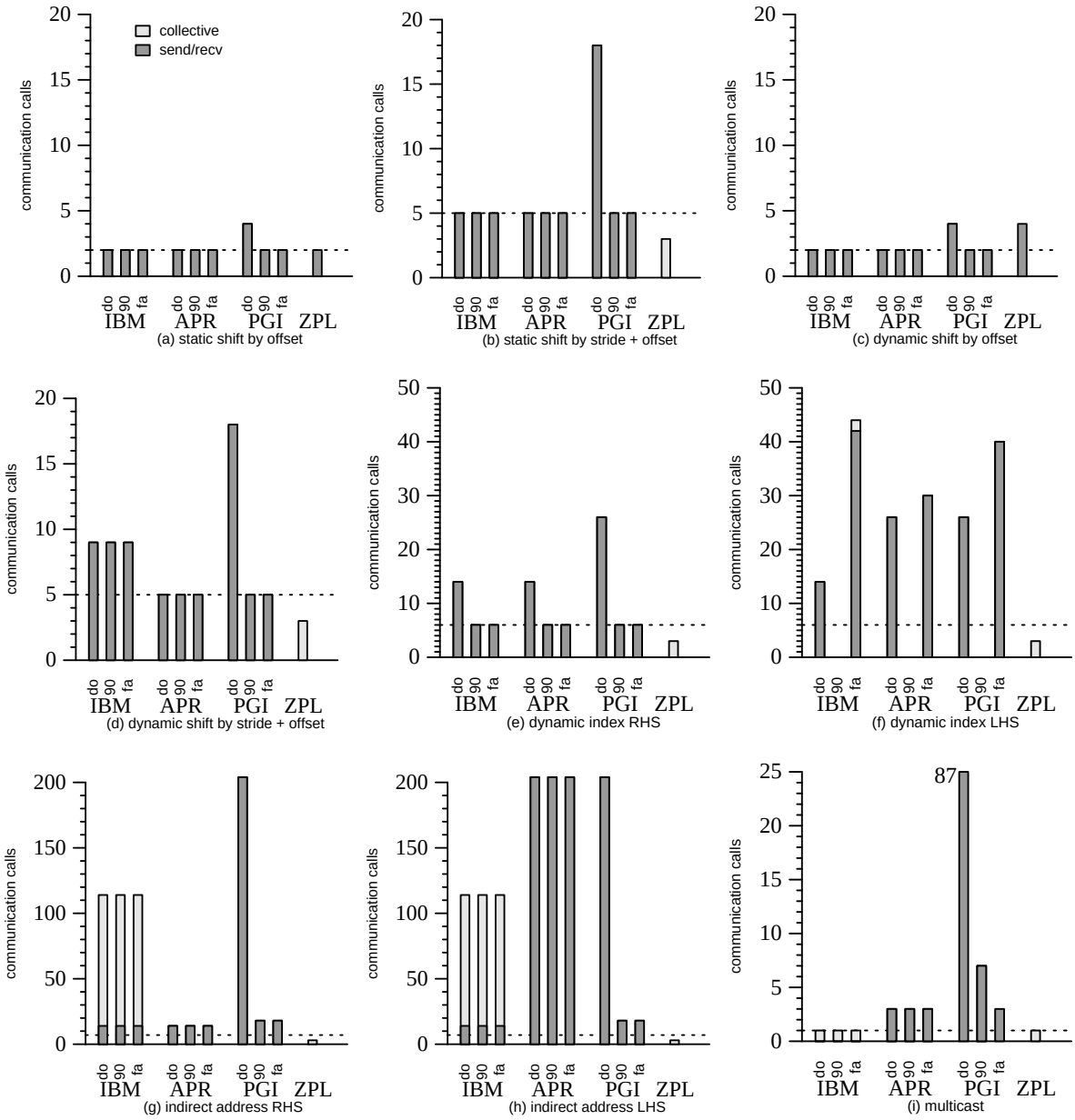


Figure 4.2: Communication for array assignment using different index expressions: p=8. Note: the dotted line represents the maximum number of sends/receives invoked if the communication is programmed manually.

typically dominates the communication time for small messages. This holds especially true for our experiment since no computation is involved, although some optimizations that overlap the communication with computation may not be effective. Although the HPF global array indexing yields no visual cue to the underlying communication, in some situations such as when the data distribution and index expression are simple, it appears possible to estimate the communication involved. It is then interesting to correlate between the user's expectation and the actual communication generated by the HPF compilers.

Since the actual count can be different on each processor, the highest count among the processors is typically used as a conservative estimate for the communication. To reflect this coarse metric, in Figure 4.2, the maximum count of communication call per processor is plotted in several dimensions: for each type of assignment, for each loop alternative, and for each compiler implementation. The dotted line represents the maximum number of sends/receives invoked per processor if the communication is programmed manually. Note that some implementations use the MPI collective communications which are composed of multiple sends/receives. For the affine index expressions, (a, b and c), the compilers generate very efficient communication. The only exception is PGI, which consistently generates excessive communication for the DO loop. For the complex index expressions, the communication schemes vary significantly both between compilers and between implementations of the same compiler.

Data dependence analysis

Figure 4.3 shows the elapsed time for each type of assignment, for each compiler, and for p=8. The time is broken down into the communication and computation components. We expect the runtime overhead to make up most of computation component since the statement itself contains no computation.

We can evaluate the quality of the data dependence analysis of the compilers by inspecting the level of performance variation across the DO, F90, and Forall implementations for each compiler. Although the loop constructs have different semantics, a

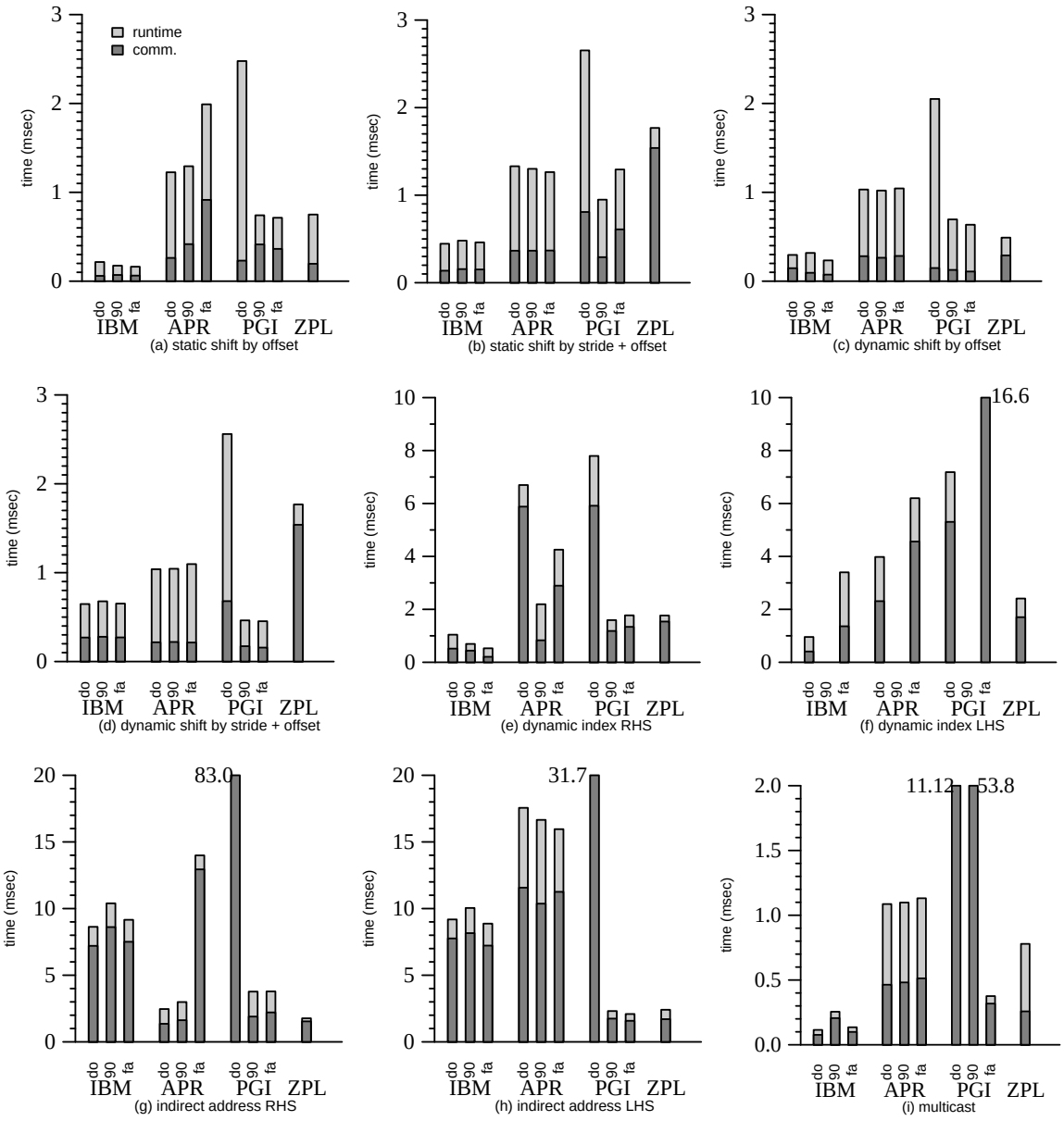


Figure 4.3: Execution time for array assignment using different index expressions: p=8

perfect analysis will be able to determine the true loop-carried dependences and optimize accordingly. Since there are no true data dependences in any of the assignments, it is possible that a compiler may yield the same performance regardless of how the assignment is expressed. In terms of the performance model, if this quality proves to be reasonably uniform across the compilers, then a user would not have to be concerned about the choice of the implementation.

Inspecting each group of DO/F90/Forall in Figure 4.3, we find that in some cases the compiler is indeed successful in its analysis, notable the IBM and APR compilers. In other cases however, variations up to 2 orders of magnitude exist between implementations of the same assignment, suggesting that data dependence analysis is not yet a common and reliable technology.

The IBM compiler shows the least performance variation, an indication of its uniform approach to loop analysis. The APR compiler shows more variation, particularly for *static shift*, *dynamic index RHS, LHS* and *multicast*, but in many cases it also shows a fairly uniform performance. The PGI compiler in general performs poorly with the DO loop, except for *dynamic index LHS* for which the DO implementation is better with all compilers, and for the *point to point* which does not involve a loop.

Note that the DO loop assignments are not annotated with the HPF $INDEPENDENT directive. This would allow the compiler to completely bypass the dependence analysis. The omission is intended to evaluate the options of parallelizing existing F77 programs and continuing to write programs in the F77 style without change. The cross-compiler variation suggests that this approach is problematic since it relies on a capability of the compiler which is not portable. The implication is thus twofold. First, parallelizing an existing sequential program will require substantially more modification than merely adding the data distribution directives: each DO loop must be analyzed for its dependences to determine whether the $INDEPENDENT directive can be used; incorrect directives will result in incorrect programs. Second, users accustomed to the sequential Fortran programming model will need to learn new semantics to write effective HPF

programs.

Runtime overhead

The runtime overhead is shown as the lighter shade in the bar graphs of Figure 4.3. This overhead typically includes the global to local index conversion, buffer allocation, data copying and any computation to determine the data distribution. The overhead is an accepted cost for a high level language, but since it is hidden from the program, a uniform overhead would greatly aid the performance model. For the IBM compiler, the overhead appears to be a fairly uniform component of the three implementations. APR's overhead is larger, but is also uniform. PGI's overhead on the other hand tends to be larger for DO loops than for F90 and Forall.

One point worth noting is that while the experiment setup has minimized most sources of variation, some differences inevitably remain in the compilers, in particular for the scalar code. Specifically, ZPL uses C as the intermediate language, APR's and PGI's HPF use Fortran as the intermediate language, while IBM's HPF is integrated with the native IBM Fortran compiler. One may expect some degradation in the scalar performance from Fortran to C, and from native to nonnative compilers.

Summarizing the case studies, we recall that HPF does not specifically provide a performance model for the array assignment. However, by making some assumptions based on the expected difficulty for the compiler in analyzing the DO loop, F90 and Forall contructs, we estimated that the Forall implementation should be the best choice in general. We find that some cases meet our expectation, but other cases do not. No single choice is consistently the best choice for all index expressions. Even when we focus on a single type of index, in some cases it is not possible to choose one implementation that is consistently the best across the compilers. For the users, this is a clear shortcoming of the performance model that will hinder the scalability, portability and ease of use of the language.

4.3 Selecting an algorithm: matrix multiplication

Earlier discussion has stated that the selection of the best algorithm for the system contributes the most to the scalability of a program. The selection in turn depends on the performance model presented by the language. In this case study, the problem to be considered is matrix multiplication. Four alternatives are considered for HPF and two for ZPL; they are described in the following subsections. The set of compilers and the parallel platform remain the same as in the preceding case study.

4.3.1 HPF versions

The four alternatives for HPF are intended to illustrate the reasoning based on the performance model to select the best solution for the problem. The code segments are shown in Table 4.3.

The most straightforward algorithm is the triply nested DO loop (Table 4.3(a)). This case shows the simplest mode of usage for HPF: a conventional sequential algorithm is annotated with HPF directives to partition the array, and the compiler parallelizes the program by distributing the data and computation. This case also shows that no communication is evident in the program; it depends entirely on the compiler to generate the necessary communication.

To improve on this algorithm, an experienced Fortran programmer may notice that the 2-D block distribution may not be a good match for the Fortran storage model which favors traversing with the leftmost index. Because the matrix traversal will be interrupted in a 2-D distribution, a 1-D distribution for CY and CA along the second dimension may allow each processor to traverse its first dimension continuously. The loops are also reordered to favor spatial and temporal locality. In addition, the *INDEPENDENT* directives are inserted in case the compiler has difficulty determining that no data dependences exist[1].

As the third alternative, a user may recognize that the targeted machine has a dis-

[1] This version was made available by David Torres, University of New Mexico.

Table 4.3: Matrix multiplication algorithms expressed in HPF.

(a) Conventional triply nested loop:

```
        real CA(M,N), CX(N,P), CY(M,P)
!HPF$   processors, dimension(2,2) :: PG
!HPF$   distribute CY(block,block) onto PG
!HPF$   distribute CA(block,block) onto PG
!HPF$   distribute CX(block,block) onto PG
        do I = 1,M
          do J = 1, P
            do K = 1, N
              CY(I,J) = CY(I,J) +
                        CA(I,K) * CX(K,J)
            enddo
          enddo
        end do
```

(b) Triply-nested loop optimized for HPF:

```
        real CA(M,N), CX(N,P), CY(M,P)
!HPF$   processors, dimension(4) :: PG
!HPF$   distribute CY(*,block) onto PG
!HPF$   distribute CA(*,block) onto PG
!HPF$   distribute CX(block,*) onto PG
!HPF$   INDEPENDENT
        do I = 1,P
          do J = 1, N
!HPF$     INDEPENDENT
            do K = 1, M
              CY(K,I) = CY(K,I) +
                        CA(K,J) * CX(J,I)
            enddo
          enddo
        enddo
```

(c) Cannon's algorithm:

```
        real CA(M,N+1), CX(N+1,P), CY(M,P)
!HPF$   processors, dimension(2,2) :: PG
!HPF$   template T(M+1,N+1)
!HPF$   distribute T(block,block) onto PG
!HPF$   align CA(i,j) with T(i,j)
!HPF$   align CX(i,j) with T(i,j)
!HPF$   align CY(i,j) with T(i,j)
        ...
        ! Multiply CY = CA * CX
        ! First skew CA and CX
        do I=2,M
          CA(I:M,N+1) = CA(I:M,1)
          CA(I:M,1:N) = CA(I:M,2:N+1)
        enddo
        do J=2,P
          CX(N+1,J:P) = CX(1,J:P)
          CX(1:N,J:P) = CX(2:N+1,J:P)
        enddo
        !then dot product and shift
        do I=1,N
          CY(1:M,1:P) = CY(1:M,1:P) +
                        CA(1:M,1:N) * CX(1:N,1:P)
          CA(1:M,N+1) = CA(1:M,1)
          CA(1:M,1:N) = CA(1:M,2:N+1)
          CX(N+1,1:P) = CX(1,1:P)
          CX(1:N,1:P) = CX(2:N+1,1:P)
        enddo
```

(d) SUMMA algorithm:

```
        real CA(M,N), CX(N,P), CY(M,P)
!HPF$   processors, dimension(2,2) :: PG
!HPF$   template T(M,N)
!HPF$   distribute T(block,block) onto PG
!HPF$   align CA(i,j) with T(i,j)
!HPF$   align CX(i,j) with T(i,j)
!HPF$   align CY(i,j) with T(i,j)
        ! Multiply CY = CA * CX
        ! spread and dot product and shift
        do I=1,N
          CY(1:M,1:P) = CY(1:M,1:P) +
              + spread(CA(1:M,I),2,N) *
                spread(CX(I,1:P),1,N)
        enddo
```

tributed memory; therefore an algorithm specifically designed for a distributed memory machine may be more appropriate. In this respect, Cannon's algorithm is a good candidate since it contains regular data motions and requires mostly array operations which can be expressed with the F90 syntax to avoid the DO loop dependences. As shown in Table 4.3(c) however, Cannon's algorithm is not an obvious choice since it requires careful data alignment and the program is more cumbersome than the triply nested loops.

Finally, the SUMMA algorithm [van de Geijn & Watts 95] has been shown to be effective on distributed memory machines. The algorithm presents a counter-intuitive advantage: although more messages are generated and their sizes are smaller, the regular pattern of communication and computation allows the implementation to be better optimized in many aspects, leading to an overall better performance. The HPF implementation in Table 4.3(d) is surprisingly simple and similar to the ZPL implementation shown in the next section. The use of the intrinsic *spread* suggests that it may be possible to predict the communication involved. If we assume that *spread* is implemented as a multicast, the actual implementation of the HPF program may accurately reproduce the intended algorithm.

4.3.2 ZPL versions

The ZPL programming model immediately throws the conventional triply nested DO loop into question. While the DO loop can be transcribed directly into ZPL using *indexed arrays*, the result is a sequential implementation. The performance model thus indicates clearly that the implementation will achieve no speedup and will have the lowest performance. To express the computation using parallel arrays, some data motion must be arranged to align the index since arithmetic operators only apply to array elements with the same index. This requirement again manifests the performance model of ZPL: the communication cost is clearly visible in the program. Given this requirement, the user begins to devise a data motion scheme to enable the element-wise arithmetic

Table 4.4: Matrix multiplication algorithms expressed in ZPL.

(a) Cannon algorithm:

```
region          RA = [1..M,1..N];
                RB = [1..N,1..P];
                RC = [1..M,1..P];
direction       east = [0,1];
                south = [1,0];
var             A: [RA] float;
                B: [RB] float;
                C: [RC] float;
                ...
                for i := 2 to M do
                    [east of RA] wrap A;
                    [i..M, 1..N] A := A@east;
                end;
                for i := 2 to P do
                    [south of RB] wrap B;
                    [1..N, i..P] B := B@south;
                end;
                for i := 1 to N do
                    C := C + A*B;
[east of RA]        wrap A;
[RA]                A := A@east;
[south of RB]       wrap B;
[RB]                B := B@south;
                end;
```

(b) SUMMA algorithm:

```
region
                RA = [1..M,1..N];
                RB = [1..N,1..P];
                RC = [1..M,1..P];
                FCol = [1..M,*];
                FRow = [*,1..P];

var             A : [RA] float;
                B : [RB] float;
                C : [RC] float;
                Aflood : [FCol] float;
                Bflood : [FRow] float;
                ...
                for i := 1 to N do
[FCol]              Aflood := >>[1..M,i] A;
[FRow]              Bflood := >>[i,1..P] B;
                    C += (Aflood * Bflood);
                end;
```

operations, and the Cannon and Summa algorithm quickly become favorable candidates. The ZPL implementations are shown Table 4.3.2. Assuming that the processor grid is $P_r \times P_c$, we can inspect the code to determine that Cannon will require approximately $M(P_r + P_c) + N(P_r + P_c)$ messages, while SUMMA will require about $N(P_r + P_c)$. In addition, Cannon has more potential synchronization. Thus the performance model indicates that SUMMA will yield better performance.

4.3.3 Results

Figure 4.4 shows the performance of each alternative algorithm in HPF and ZPL on the SP2 for a 2000×2000 matrix multiplication on 16 processors. For HPF, the performance is plotted across a compiler dimension that includes the IBM, APR and PGI compilers.

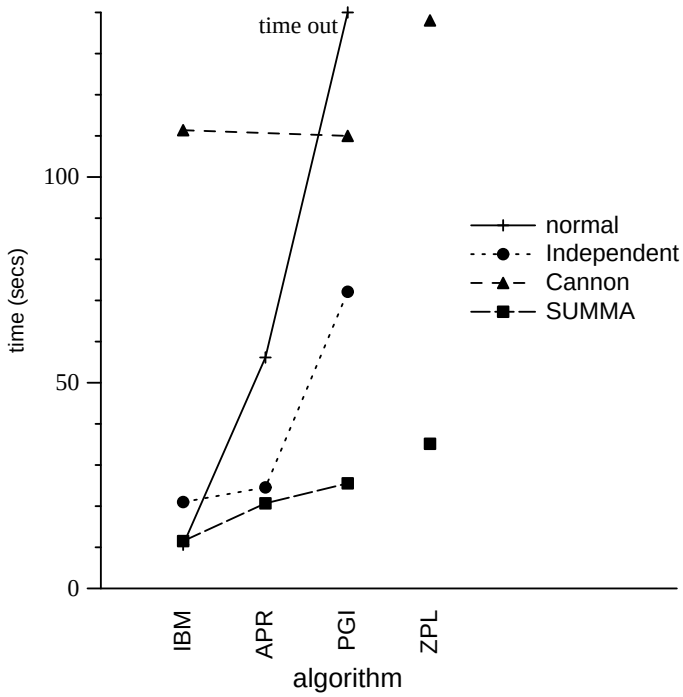


Figure 4.4: Matrix Multiplications by ZPL and 3 HPF compilers: 2000x2000, p=16

Returning to the methodology outlined in section 4.1.4, we inspect the performance for the alternatives and quickly find that equation 4.1 does not hold: they show performance differences of at least an order of magnitude. For equation 4.2, the requirement $\underline{r} = \underline{r'}$ would be indicated by performance lines that maintain the same relative order and that do not intersect each other. Again, a quick inspection of Figure 4.4 reveals that the line for the triply nested DO loop intersects all other lines. Interestingly for HPF, the relation $\underline{f}$ in equation 4.2 is not completely undefined in this case: the description of the algorithm in the previous section indicates that some ordering can be expected based on a number of assumptions. We now consider each case in more detail by examining the performance and the implementation. Table 4.5 shows the pseudo-code for the implementations generated by the HPF and ZPL compilers. Recall that the basic effect of parallelization is the adjustment of the loop index to span the local partition and the

Table 4.5: Pseudo-code for the matrix multiplication algorithms by three HPF compilers.

Note: (1) For the loops, *full* means all iterations while *part* means local partition only
(2) Cannon is not shown because all implementations are virtually identical.

version	IBM	APR	PGI	ZPL

3-nested DO loop	comm(ca)
	comm(cx)
	do i=part
	  do j=part
	  do k=full
	  cy=cy+ca*cx
	  enddo
	  enddo
	enddo
		comm()
		do i=full
		  do j=part
		  do k=full
		  cy=cy+ca*cx
		  enddo
		  enddo
		enddo
		comm()
			do i=full
			  do j=full
			  do k=full
			  comm(ca)
			  comm(cx)
			  cy=cy+ca*cx
			  enddo
			  enddo
			enddo
				n/a

HPF-opt	comm(ca)
	comm(cx)
	do i=part
	  do j=full
	  do k=full
	  cy=cy+ca*cx
	  enddo
	  enddo
	enddo
		comm()
		do i=part
		  do j=full
		  do k=full
		  cy=cy+ca*cx
		  enddo
		  enddo
		enddo
		comm()
			comm(cy)
			comm(cx)
			comm(ca)
			do i=part
			  do j=full
			  do k=part
			  cy=cy+ca*cx
			  enddo
			  enddo
			enddo
			comm(cy)
				n/a

SUMMA	comm(ca)
	comm(cx)
	do i=full
	  do j=part
	  do k=part
	  cy=cy+ca*cx
	  enddo
	  enddo
	enddo
		comm()
		do i=part
		  do j=full
		  do k=full
		  cy=cy+ca*cx
		  enddo
		  enddo
		enddo
		comm()
			do i=full
			  comm(ca)
			  comm(cx)
			  do j=part
			  do k=part
			  cy=cy+ca*cx
			  enddo
			  enddo
			enddo
				do i=full
				  comm(ca)
				  comm(cx)
				  do j=part
				  do k=part
				  cy=cy+ca*cx
				  enddo
				  enddo
				enddo

insertion of communication at the appropriate point. The loops in Table 4.5 that are parallelized are identified as *part*, while those not parallelized are *full*. Communication placed outside of the loops results in fewer messages and less overhead.

DO loops

Clearly, no expectation is possible for the triply nested DO loop regarding the implementation or the performance, and this is evident in the data. The implementations vary widely between the HPF compilers in the placement of communication and computation. This wide variation can be expected given that by design the compilers are free to generate any implementation; therefore the issue is not how the compilers arrive at their implementation, but whether it is possible for the performance behavior to be consistent.

The triply nested DO loop does not yield portable performance: the performance ranges from very high for IBM to very low for PGI. The IBM implementation places all communication at the beginning, followed by the loops. This enables the standard IBM Fortran compiler to perform very effective loop transformations to optimize the code for the superscalar CPU (instruction scheduling, pipelining); as a result, the IBM version enjoys very high scalar performance. The PGI implementation on the other hand uses a guard to determine if a processor is to perform the computation; therefore although the actual computation is distributed, the scheme requires each processor to scan all N^3 iterations. In addition, the communication remains in the inner most loop, resulting in excessive communication.

HPF-optimized DO loops

The optimization for this version has been made based on a number of assumptions. If they prove to be correct, then we can establish the relation $\underline{f}$ in equation 4.2 between the optimized and the standard DO loop versions. Our interest in the implementation is whether the tuning of the program has the desired effect. However, no expectation is guaranteed by HPF.

The PGI compiler elects to depart from the owner-computes rule and redistributes

the computation into 2-D instead of the specified 1-D, apparently because it detects that two *INDEPENDENT* loops exist in the program. This requires additional communication before the loops to move the LHS to the site of computation and after the loops to return the LHS to the original distribution. Recall that the user optimization strategy for this program assumes the owner-computes rule; otherwise it is not possible to reason about the access patterns. Yet, because the compiler is free to distribute the computation, we find that there is little correlation between the actual implementation and the expectation.

With respect to performance, APR and PGI improve by a factor of 2 or more, while IBM's performance degrades by a factor of 2 to 4. The results thus contradict all aspects of equation 4.2.

Cannon's algorithm

All implementations for Cannon are similar and they are consistent with the algorithm. For this reason and because they are more verbose, they are omitted from Table 4.5. Examining the implementations, the consistency stems from the treatment of the array statement and the use of a particular index expression in HPF. The array semantics call for the RHS to be read completely before the assignment is made to the LHS; all compilers adhere to this semantics by fetching the RHS before and storing to the LHS after the statement. In addition, the HPF version only requires an index expression of the type *static shift*, which by its simplicity has been shown to exhibit good performance behavior (Section 4.2). With respect to the implementation, Cannon thus represents an instance where the requirement of the algorithm matches the semantics and the performance model of both HPF and ZPL.

However, the performance for Cannon does not meet the expectation for an algorithm designed for distributed memory machines. All Cannon implementations suffer from the high overhead of repeated buffer allocation and deallocation for each message, data copying and conservative synchronization for each communication phase. APR in particular performs quite poorly. Given that the performance model for HPF and ZPL

is successful in this case, further compiler optimization is needed to reduce the runtime overhead and to deliver the expected performance.

The ZPL performance is slightly lower than the HPF performance. On examining the assembly code, we find that the C compiler for ZPL intermediate code is not as effective as the Fortran compiler in instruction scheduling and pipelining.

SUMMA

For each outer loop iteration, the SUMMA algorithm calls for a row and a column to be replicated across all processors in the first and second dimension of the processor grid, respectively. The intrinsic *spread* offers this functionality; therefore its use suggests that the program as expressed will implement the SUMMA algorithm faithfully.

We find that the IBM compiler chooses to first scalarize *spread* into two inner DO loops, then optimize all three DO loops together. This yields an interesting result: the IBM implementation for SUMMA is very similar to the triply nested DO loop implementation in which all communication occurs at the beginning, outside the outermost loop (Table 4.5). Although IBM's optimization strategy appears to be sound for the general case, in this case it has in effect unintentionally transformed one algorithm into another algorithm that is potentially less optimal in the larger parameter space not explored here[van de Geijn & Watts 95].

The APR implementation also does not reflect the SUMMA algorithm. APR consistently partitions along one dimension regardless of the specified distribution. Consequently, APR's implementations of all algorithms follow a similar scheme: RHS values are fetched before the loop and LHS values are sent to the owner after the loop. In this case, the communication occurs outside of the outermost loop. IBM and APR thus prove our assumption regarding *spread* to be incorrect. HPF does not specify how the intrinsic is to be implemented, and again the implementation does not match the intended algorithm.

The implementation by PGI and ZPL accurately reproduces the SUMMA algorithm. Based on the different approaches taken by IBM and APR, PGI's successful correlation

Table 4.6: Speedup ratio from 16 to 64 processors, 2000 $\times$ 2000 matrix multiplication. Note: (1) execution timed out or memory exhausted.

speedup	normal	HPF-optimized	cannon	summa
IBM	3.44	1.46	2.98	2.72
APR	0.94	1.34	(1)	2.60
PGI	(1)	5.58	2.94	3.03
ZPL	n/a	n/a	3.08	3.49

in this case is incidental rather than by design. The communication in the ZPL implementation on the other hand correlates directly with the >> operator in the program as specified by the performance model.

Interestingly, the SUMMA performance by all compilers is consistently the best compared to the other algorithms. IBM and APR in particular achieve good performance although their implementations do not reflect SUMMA. SUMMA has been shown to be superior in many aspects such as memory usage, generality and flexibility for non-square matrices [van de Geijn & Watts 95]. Our limited experiment in this case does not cover the parameter space sufficiently to illustrate the advantage of SUMMA. Therefore, the good performance belies the fact that the actual implementations do not reflect the same algorithm. The ZPL performance SUMMA is lower than the HPF's performance because of the scalar component. An inspection of the assembly codes generated by the Fortran and C compilers shows that the Fortran compiler can generate a much better instruction schedule for pipelining the superscalar processor in the SP2.

Finally, we consider the self-relative speedup from 16 to 64 processors (Table 4.6). Compared to the ideal speedup of 4, we find that APR and PGI fail to achieve any speedup for the DO loop version; APR's Cannon also fails because of high memory allocation. IBM and APR only achieve a modest speedup when the DO loop version is optimized, while PGI achieves superlinear speedup. This unpredictable variation in the

speedup is another indication of the weakness of HPF's performance model. In contrast, both ZPL implementations scale well.

In summary, we find that little correlation exists between the algorithm expressed in the HPF programs and the actual implementations. By HPF's design, the compiler has significant freedom in transforming the program to arrive at an implementation. This characteristic in effect constitutes a gap in the performance model that essentially prevents the users from making any performance prediction for an algorithm. In contrast, ZPL correctly predicts that SUMMA is a better choice than Cannon, which in turn is better than the triply nested DO loop (not implemented). ZPL thus satisfies equation 4.2 in each case by matching the relative performance predicted by the performance model with the relative performance of the actual implementations.

4.4 Current HPF solutions

A consequence of the weak performance model in HPF is the general difficulty in predicting the behavior of an HPF program. This is evident in the solutions offered by software vendors or features that programmers rely on for tuning HPF programs.

A part of the HPF compiler from APR is an extensive set of compiler options and tools for profiling and analyzing the program performance. The program can be automatically instrumented to gather statistics at each DO loop level and the statistics can serve as a database to the compiler for further recompilation and optimization. The listing generated by the compiler also provides performance details not available in the source program such as which loop level is being parallelized, which statement may incur communication, where communication is inserted and which array is being communicated. In addition, the FORGE Explorer Distributed Memory Parallelizer allows a user to interactively choose the arrays to partition and the loops to distribute. The user can also insert additional APR directives to aid the compiler in parallelizing loops or reducing communication.

The capability of the APR system may sufficiently supplement the HPF performance

model so that a user can achieve good performance with the system. Indeed, users proficient with the FORGE system may have learned a fairly complete model for producing good parallel programs. The success of this approach is evident in the benchmarks that APR made publicly available in conjunction with their published HPF performance; the benchmarks are highly tuned to the APR compiler, containing liberal APR specific directives to aid the compiler. Unfortunately, such a model is a superset rather than a part of the HPF language specification. It is not portable to other HPF compilers, and there is no evidence that it should be formalized as HPF's model.

Another program tuning technique that is more generally accessible is the intermediate SPMD code generated by the compilers. This option is available in all three HPF compilers being studied, even though the IBM compiler is a native compiler and does not need to generate the intermediate output. The intermediate code can be difficult to decipher, but has proven indispensable in providing important clues such as the communication generated for a particular statement.

4.5 Conclusions

In this chapter, we identify the performance model as a crucial component of the language that programmers rely on for selecting the best implementation or algorithm. Because HPF and ZPL are data parallel languages for distributed memory parallel machines, the relevant aspects of the performance model are the communication and the overhead for managing the data distribution. To quantify the benefit of the performance model, we formulate a framework which leads to two requirements.

For a language with no performance model, equation 4.1 requires the compiler to neutralize any performance difference between two alternative programs because the programmer has no means to make a selection.

For a language to have a performance model, equation 4.2 requires that the relative performance predicted by the model matches the implementation.

Two case studies are performed using the array assignment and the matrix multi-

plication. In each case, HPF fails to satisfy both of the requirements above, while ZPL consistently meets the requirements. Since the performance model is needed to select between alternatives, the potential cost or benefit of the model is equivalent to the performance difference between the alternatives. The data from both studies shows that this performance difference is at least one order of magnitude. This result indicates that HPF users will face difficulties in achieving consistent and portable performance in their programs.

In practice, a performance model can be extended beyond the language specification. A user accustomed to a particular HPF compiler will over time learn the behavior of the compiler and supplement any missing information in the HPF performance model with compiler specific information. For instance, the user may assume that F90 array assignment will be the most efficient. The supplemented model may enable the user to write high performance programs using the particular compiler, but since this model is not portable across compilers and consequently across platforms, the program performance will not be portable.

It appears that part of HPF problem lies in the multiple alternatives for expressing the parallel computation. One may conjecture whether HPF can limit the number of alternatives to present a more consistent model. However, our discussion shows that a performance model requires a more careful effort than simply limiting the alternatives. Furthermore, this conflicts with the goal of compatibility with Fortran since the DO loop is an integral part of Fortran 77 as the array syntax is an integral part of Fortran 90. At the same time, the new Forall construct is desirable since the DO loop is difficult to analyze while the F90 syntax is too restrictive.

In summary, this chapter demonstrates the importance of the performance model. The detailed study of the implementations shows that to create a robust performance model, the language specification must be sufficiently concise for the programmer to rely on and for the compiler to implement with consistency.

Chapter 5

Benchmark Comparison

5.1 Introduction

Recently, several commercial HPF compilers have become available, enabling users to learn HPF, program in it, and directly evaluate the language. The Cornell Supercomputer Center has made available to the scientific community a comprehensive set of HPF compilers and tools for production use. A ZPL compiler developed at the University of Washington is also publicly available for several parallel platforms, including the KSR, Intel Paragon, IBM SP2, and Cray T3D. Since no broad evaluation of the language and the compilers is yet available, this chapter is focused on the effectiveness of HPF and ZPL in achieving portable and scalable performance for data parallel applications.

We will study in-depth the performance of three NAS benchmarks compiled with three commercial HPF compilers on the IBM SP2. The benchmarks are: Embarrassingly Parallel (EP), Multigrid (MG), and Fourier Transform (FT). The HPF compilers include Applied Parallel Research, Portland Group, and IBM. To evaluate the effect of data dependences on compiler analysis, we consider two versions of each benchmark: one programmed using DO loops, and the second using F90 constructs and/or HPF's Forall statement. The ZPL compiler is a version ported to the IBM SP2. For comparison, we also consider the performance of each benchmark written in Fortran with MPI.

The MPI results represent a level of performance that the HPF program should target to be considered a viable alternative. The ZPL version gives an indication of the performance achievable when the compiler is not hampered by language features unrelated to parallel computation.

The results show some successes with the F90/Forall programs but the results are not uniform. For the other programs, the results suggest that Fortran's sequential nature causes considerable difficulty for the compiler's analysis and optimization of the communication. The varying degrees of success among the compilers in parallelizing the programs, coupled with the absence of a clear model to guide the insertion of directives, results in an uncertain programming model for users as well as portability problems for HPF programs.

In related work, APR published the performance of its HPF compiler for a suite of HPF programs, along with detailed descriptions of their program restructuring process using the APR FORGE tool to improve the codes [App 95, Friedman et al. 95]. The programs are well tuned to the APR compiler and in many cases rely on the use of APR-specific directives rather than standard HPF directives. Although the approach that APR advocates (program development followed by profiler-based program restructuring) is successful for these instances, the resulting programs may not be portable with respect to performance, particularly in cases that employ APR directives. Therefore, we believe that the suite of APR benchmarks is not well suited for evaluating HPF compilers in general.

Similarly, papers by vendors describing their individual HPF compilers typically show some performance numbers, but the benchmarks tend to be selected to highlight the specific compiler's strengths. Consequently, it is difficult to perform comparisons across compilers [Harris et al. 95, Bozkus et al. 95, Gupta et al. 95].

Lin *et al.* used the APR benchmark suite to compare the performance of ZPL versions of the programs against the corresponding HPF performance published by APR and found that ZPL generally outperforms HPF. However, the lack of access to the

APR compiler did not allow detailed analysis, limiting the comparison to the aggregate timings [Lin et al. 95].

The goals in this chapter are:

1. An in-depth comparison and analysis of the performance of HPF programs with three current HPF compilers and alternative languages (MPI, ZPL).
2. A comparison of the DO loop with the F90 array syntax and the Forall construct.
3. A comparison of machine-specific and portable HPF compilers.
4. An assessment of the parallel programming model presented by HPF.

The chapter is organized as follows: Section 5.2 describes the methodology for the study, including a description of the algorithms and the benchmark implementations. In Section 5.3, we examine and analyze the benchmarks' performance, detailing the communication generated in each implementation and quantifying the effects of data dependences in the HPF programs. Section 5.4 provides our observations and our conclusions.

5.2 Methodology

5.2.1 Overall approach

We study the NAS benchmarks EP, MG and FT [Bailey et al. 91]. MG and FT are derived from the NAS benchmark version 2.1 (NPB2.1), published by the Numerical Aerodynamic Simulation group at NASA Ames [Bailey et al. 95]. Previous versions of the NAS benchmarks were implemented by computer vendors and were intended to measure the best performance possible on a parallel machine without regard to portability. NPB2.1 is an MPI implementation by NAS and is intended to measure the best *portable* performance for an application. Because NPB2.1 programs are portable and are originally parallel, they constitute an ideal base for our study. Specifically, NPB2.1 serves two purposes:

1. The actual NPB2.1 performance serves as the reference point for the HPF implementations or for any high level language [Saphir et al. 95].

2. The HPF implementations are derived by *reverse engineering* the MPI programs: communication calls are removed and the local loop bounds are replaced with global loop bounds. HPF directives are then added to parallelize the programs. Conceptually, the task for the HPF compilers is to repartition the problem as specified by the HPF directives and to regenerate the communication.

Since EP is not available in NPB2.1, we use the version from the benchmark suite published by APR.

An important issue in an HPF program is how the computation is expressed. It was recognized that the conventional Fortran DO loop may over-specify the data dependences in data parallel computations; therefore the Fortran 90 array syntax and the Forall construct were proposed as better alternatives for expressing parallelism [Forum 93]. Since the NPB2.1 programs (and the derived HPF programs) are written in Fortran 77 with DO loops, we also consider a version in which the DO loops for whole-array operations are replaced with Fortran 90 syntax or the HPF Forall construct.

To focus on the portability issue, the HPF directives are used according to the HPF specification within the functionality limit of the compilers; in other words, the programs are not tuned to any specific compiler. This may put the APR compiler at a disadvantage since it relies on the APR-provided profiling and program restructuring tools. The chosen benchmarks use only the basic HPF intrinsics and require only the basic BLOCK distribution that is supported by all three compilers. Therefore they can stress the compilers without exceeding their capability.

The implementations in the ZPL language are derived from the same source as the HPF implementations, but in the following manner: the sequential computation is translated directly from Fortran to the corresponding ZPL syntax, while the parallel execution is expressed using ZPL's parallel constructs.

5.2.2 Benchmark selection

NPB2.1 contains 7 benchmarks[1], all of which should ideally be included in the study. Unfortunately, a *portable* HPF version of these benchmarks is not available, severely limiting an independent comparison. While APR and other HPF vendors publish the benchmarks used to acquire their performance measurements, these benchmarks are generally tuned to the specific compiler and are not portable. This limitation forces us to carefully derive HPF versions from existing benchmarks with the focus on portability while avoiding external effects such as algorithmic differences. The following criteria are used:

1. The benchmarks should be derived from an independent source to insure objectivity.

2. A message passing version should be included in the study since the comparison is not only between HPF and ZPL but also against the target that HPF and ZPL are to achieve.

3. For HPF, there should be separate versions that employ F77 DO loop and F90/Forall because there is a significant difference between the two types of construct.

4. There should be no algorithmic differences between versions of the same benchmark.

5. Tuning must adhere to the language specification rather than any specific compiler capability.

6. Because support for HPF features is not uniform, the benchmarks should not require any feature that is not supported by all HPF compilers.

Considering the benchmark availability in Table 5.1, the sources for NPB1 are generally sequential implementations. Although they could be valid HPF programs, the

[1] One was added recently after this writing

sequential nature of the algorithms may be too difficult for the compilers to parallelize and may not reflect a natural approach to parallel programming. In other words, an HPF programmer may have chosen a specific parallel algorithm and simply wants to implement it in HPF. APR's sources, as mentioned above, are tuned to the APR compiler; therefore they are not appropriate as a portable implementation. The NPB2.1 sources are the best choice since they implement inherently parallel algorithms and they use the same MPI interface as all compilers being studied.

Among the benchmarks, CG is eliminated simply because it is not available in NPB2.1. SP, BT and LU cannot be included because they require a block cyclic and 3-D data distribution that is not supported by all HPF compilers and ZPL. The limitations in themselves do not prevent these benchmarks to be implemented in HPF and ZPL (indeed they are); however the implementations will have an algorithmic difference that cannot be factored from the performance. This leaves only FT and MG as potential candidates. Fortunately, EP is by definition highly parallel; therefore its sequential implementation can be trivially parallelized.

5.2.3 Platform

The targeted parallel platform is the IBM SP2 at the Cornell Theory Center. The system is a distributed memory machine with 512 processors, 48 of which are wide nodes[2]. The compilers used in the study include:

- Portland Group pghpf version 2.1
- IBM xlhpf version 1.0
- Applied Parallel Research xhpf version 2.0
- ZPL compiler (SP2 port) from the University of Washington

One potential source of difference between ZPL and HPF performance is that the ZPL compiler produces C as the intermediate code. In general, Fortran compilers have

[2] Wide SP2 nodes have wider data path and larger caches.

Table 5.1: Sources of NAS benchmarks for HPF and ZPL
The left 3 columns show the availability of the sources; the right 3 columns show the versions needed for the study and the source used.

	Available			Needed for study		
	NPB1	tuned APR	NPB2.1	HPF DO	HPF F90	ZPL
EP	yes	yes	N/A	tuned APR	NPB1	NPB1
FT	yes	yes	yes	NPB2.1	NPB2.1	NPB2.1
CG	yes	N/A	N/A	N/A	N/A	N/A
MG	yes	yes	yes	NPB2.1	NPB2.1	NPB2.1
SP	yes	yes	yes	N/A	N/A	NPB1
BT	yes	yes	yes	N/A	N/A	N/A
LU	yes	N/A	yes	N/A	N/A	N/A

more opportunities for scalar optimization than a C compiler. To obtain a coarse approximation of this difference, we converted the FT benchmark from Fortran to C and compared the performance. For 1 to 8 processors, the Fortran version is 32% to 42% faster than the C version. This suggests that the observed scalar performance for ZPL will tend to be conservative.

Processors	Fortran	C	% Δ over C
1	4.70 secs	7.55 secs	38%
4	1.49 secs	2.20 secs	32%
8	.83 secs	1.43 secs	42%

All compilers generate MPI calls for the communication and use the same MPI library, ensuring that the communication fabric is identical for all measurements. Besides the aggregate timings, the program execution is also traced using the UTE facility [IBM 95] to measure the major phases of the program and the communication.

All measurements use the same compiler options and system environment that NAS and APR specified in their publications, and spot checks confirmed that the published NAS and APR performances are reproduced in this computing environment. The following sections will briefly describe the benchmarks and give further details on how the HPF (DO loop and F90/Forall) and ZPL implementations were created.

5.2.4 Embarrassingly Parallel

The benchmark EP generates N pairs of pseudo-random floating point values (x_j, y_j) in the interval (0,1) according to the specified algorithm, then redistributes each value x_j and y_j onto the range (-1,1) by scaling them as $2x_j - 1$ and $2y_j - 1$. Each pair is tested for the condition:

$$t_j \leq 1 \;\; where \; t_j = x_j^2 + y_j^2$$

If true, the independent Gaussian deviates are computed:

$$X = x_j \sqrt{(-2 \log t_j)/t_j}$$

$$Y = y_j \sqrt{(-2 \log t_j)/t_j}$$

Then the new pair (X, Y) is tested to see if it falls within one of the 10 square annuli and a total count is tabulated for each annulus.

$$l \leq max(|X|, |Y|) < l + 1 \;\; where \; 0 \leq l < 9$$

The pseudo-random numbers are generated according to the following linear congruential recursion:

$$x_k = a x_{k-1} \bmod 2^{46} \;\; where \; a = 5^{15}, x_0 = 271828183$$

The values in a pair (x_j, y_j) are consecutive values of the recursion. To scale to the (0,1) range, the value x_k is divided by 2^{46}.

Figure 5.1 illustrates the data structure, the general flow of the computation and the pseudo-codes. Clearly, the computation for each pair of Gaussian deviates can proceed

independently. Each processor would maintain its own counts of the Gaussian deviates and communicate at the end to obtain the global sum. The random number generation, however, presents a challenge. There are two ways to compute a random value x_k:

1. x_k can be computed quickly from the preceding value x_{k-1} using only one multiplication and one *mod* operation, leading to a complexity of $O(n)$. However, the major drawback is the true data dependence on the value x_{k-1}.

2. x_k can be computed independently using k and the defined values of a and x_0. This will result in an overall complexity of $O(n^2)$. Fortunately, the property of the *mod* operation allows x_k to be computed in $O(\log k)$ steps by using a binary exponentiation algorithm [Bailey et al. 91].

The goal then is to balance between method (1) and (2) to achieve parallelism while maintaining the $O(n)$ cost. Because EP is not available in the NPB2.1 suite, we use the implementation provided by APR as the DO loop version. This version is structured to achieve the balance between (1) and (2) by batching (see Figure 5.1(b)): the random values are generated in one sequential batch at a time and saved; the seed of the batch is computed using the more expensive method (2), and the remaining values are computed using the less expensive method (1). A DO loop then iterates to compute the number of batches required, and this constitutes the opportunity for parallel execution.

The F90/Forall version is derived from the DO loop version with the following modifications (Figure 5.1(c)):

- All variables in the main DO loop that cause an output dependence are expanded into arrays of the size of the loop iteration. In other words, the output dependence is eliminated by essentially renaming the variables so that the computation can be expressed in a fully data parallel manner. Since the iteration count is just the number of sequential batches, the expansion is not excessive.

- Directives are added to partition the arrays onto a 1-D processor grid.

- The DO loop for the final summation is also recoded using the HPF reduction intrinsic.

A complication arises involving the subroutine call within the Forall loop, which must be free of side effects in order for the loop to be distributed. Some slight code rearrangement was done to remove a side effect in the original subroutine, then the PURE directives were added to assert freedom from side effects. Unfortunately, support for PURE varies among the compilers. For instance, APR does not support the PURE and INTENT directives apparently because it performs interprocedural analysis to detect the side effects. APR and PGI do not allow a function to return an array, thus precluding an implementation similar to the ZPL implementation.

The ZPL version is translated in a straightforward manner from the DO loop version. The only notable difference is the use of the ZPL *region* construct to express the independent batch computation (Figure 5.1(d)).

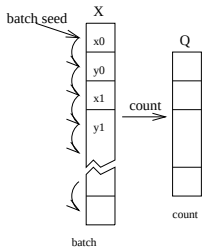
5.2.5 Multigrid

Multigrid is interesting for several reasons.

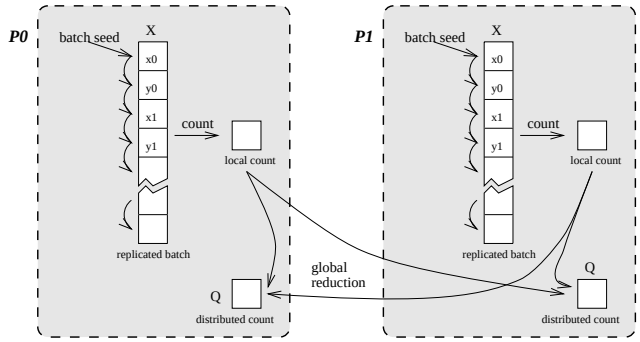
First, it illustrates the need for data parallel languages such as HPF or ZPL. The NPB2.1 implementation contains over 700 lines of code for the communication – about 30% of the program – which are eliminated when the program is written in a data parallel language.

Second, since the main computation is a 27-points stencil, the reference pattern that requires communication is simply a shift by a constant, which results in a simple neighbor exchange in the processor grid. All compilers (ZPL and HPF) recognize this pattern well and employ optimizations such as message vectorization and storage preallocation for the nonlocal data [App 95, Gupta et al. 95, Chamberlain et al. 95, Bozkus et al. 95]. Therefore, although the benchmark is rather complex, the initial indication is that both HPF and ZPL should be able to produce efficient parallel programs.

The benchmark is a V-cycle multigrid algorithm for computing an approximate so-



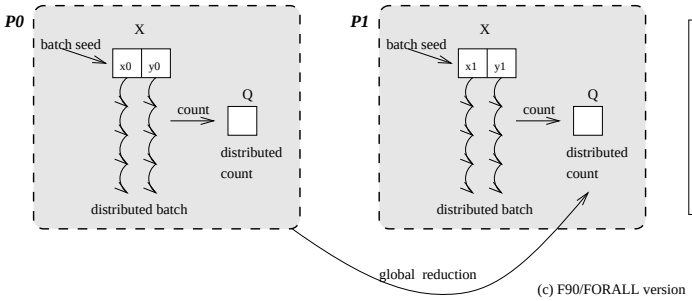
(a) NPB1 sequential version



pseudo code:

```
(1)   !HPF$ INDEPENDENT
(2)   DO  1:NB
(3)     DO
(4)       starting seed by binary expoentiation
(5)     END
(6)     CALL vranlc()  generate remaining batch
(7)     DO  1:BS
(8)       compute Gaussian deviates
(9)       tally count in concentric square annuli
(10)    END
(11)  END
```

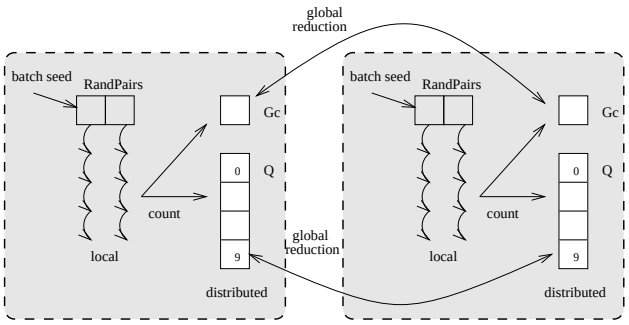
(b) Tuned APR version



pseudo code:

```
(1)   FORALL  1:NB
(2)     compute NB starting seed
(3)   DO 1:BS
(4)     X(1:NB)  =  NB next pair
(5)     X(1:NB)  = NB Gaussian deviates
(6)     tally count in concentric square annuli
(7)   END
(8)   sum(Q)
```

(c) F90/FORALL version




```
(1)   region B = [1..NB];
(2)   Gc, Q:  [B] integer;
(3)   [B] begin
(4)     Gc := RandPairs(..,Q);
(5)     for i:=0 to 9
(6)       q[i] := +\ Q[i];
(7)     gc := +\Gc;
(8)   end;
```

(d) ZPL version

Figure 5.1: Illustrations of EP as implemented in HPF and ZPL.

Note: the pseudo-codes and the data structures distributed onto two processors.

lution to the discrete Poisson problem:

$$\nabla^2 u = v$$

where ∇^2 is the Laplacian operator $\nabla^2 u = (\frac{\delta^2}{\delta x^2}, \frac{\delta^2}{\delta y^2}, \frac{\delta^2}{\delta z^2})$

The algorithm consists of 4 iterations of the following three steps:

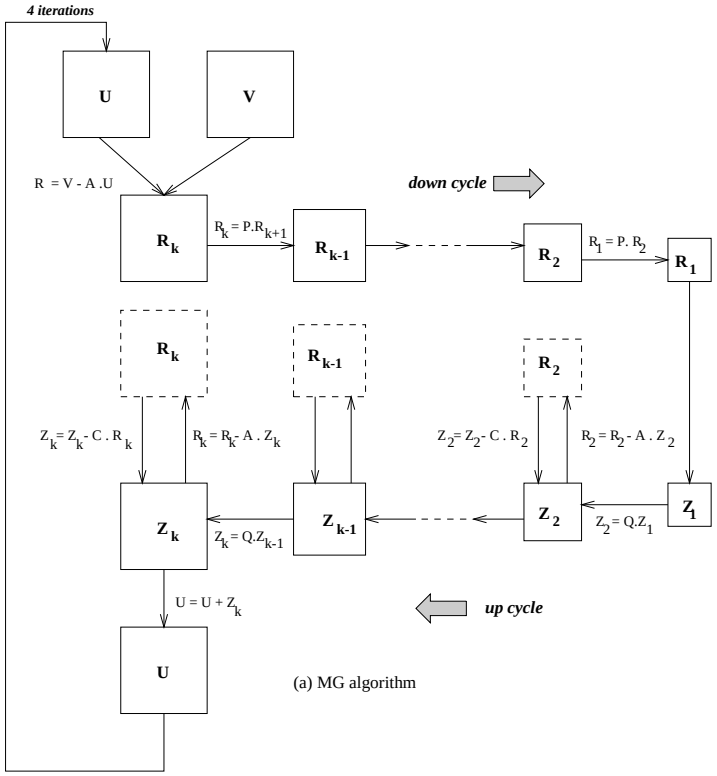
$$\begin{aligned} r &= v - A \cdot u \quad (1) \quad \textit{evaluate residual} \\ z &= M^k r \quad\quad (2) \quad \textit{compute correction} \\ u &= u + z \quad\quad (3) \quad \textit{apply correction} \end{aligned}$$

where A is the trilinear finite element discretization of the Laplace operator ∇^2, M^k is the V-cycle multigrid operator as defined in the NPB1 benchmark specification (section 2.2.2). Figure 5.2 illustrates these three steps together with the data structures: the down cycle and up cycle constitute step (2), *compute correction*. The interpolation and projection of the hierarchical grids during the down and up cycle are also illustrated for the 2-D case; note that the actual arrays are 3-D. For further details, the reader is referred to the NPB1 specification [Bailey et al. 91] as well as other publications on the vendor implementations [Agarwal et al. 95].

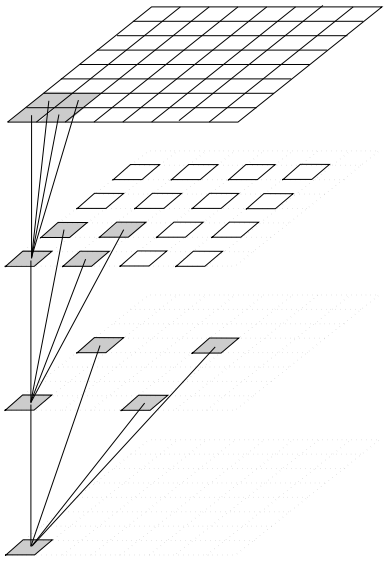
The algorithm implemented in the NPB2.1 version consists of three phases: the first phase computes the residual, the second phase is a set of steps that applies the M^k operator to compute the correction while the last phase applies the correction.

The HPF DO loop version is derived from the NPB2.1 implementation as follows:

- The MPI calls are removed.
- The local loop bounds are replaced with the global bounds.
- The use of a COMMON block of storage to hold a set of arrays of different sizes is incompatible with HPF; therefore the arrays are renamed and declared explicitly.
- HPF directives are added to partition the arrays onto a 3-D processor grid. The array distribution is maintained across subroutine calls by using the transcriptive directives to prevent unnecessary redistribution.



(a) MG algorithm



(b) Stencil computation for hierarchical grid

Figure 5.2: Illustrations for the Multigrid algorithm.

Note: (a) The arrays U, V, R_k, Z_k are full sized 3-d arrays; R_i, Z_i for $k-1 \leq i \leq 1$ are hierarchically scaled arrays. (b) This is a 2-D example of the multigrid interpolation and projection; the actual computation is 3-D.

The HPF F90/Forall version requires the additional step of rewriting all data parallel loops in F90 syntax.

The ZPL version has a similar structure to the HPF F90/Forall version, the notable difference being the use of *strided region* to express the hierarchy of 3-D grids. A strided region is a sparse index set over which data can be declared and computation can be specified.

5.2.6 Fourier Transform

Consider the partial differential equation for a point x in 3-D space:

$$\frac{\delta u(x,t)}{\delta t} = \alpha \nabla^2 u(x,t)$$

The FT benchmark solves the PDE by (1) computing the forward 3-D Fourier Transform of $u(x,0)$, (2) multiplying the result by a set of exponential values, and (3) computing the inverse 3-D Fourier Transform. The problem statement requires 6 solutions, therefore the benchmark consists of 1 forward FFT and 6 pairs of dot products and inverse FFTs.

The NPB2.1 implementation follows a standard parallelization scheme, illustrated in Figure 5.3 [Bailey et al. 95, Agarwal et al. 94a]. The 3-D FFT computation consists of traversing and applying the 1-D FFT along each dimension. The 3-D array is partitioned along the third dimension to allow each processor to independently carry out the 1-D FFT along the first and second dimension. Then the array is transposed to enable the traversal of the third dimension. The transpose operation constitutes most of the communication in the program. Note that the program requires moving the third dimension to the first dimension in the transpose so that the memory stride is favorable for the 1-D FFT; therefore the HPF *REDISTRIBUTE* function alone is not sufficient[3].

The HPF DO loop implementation is derived with the following modifications:

1. HPF directives are added to distribute the arrays along the appropriate dimension. Transcriptive directives are used at subroutine boundaries to prevent unnecessary redistribution.

[3]HPF data distribution specifies the partition to processor mapping, not the memory layout.

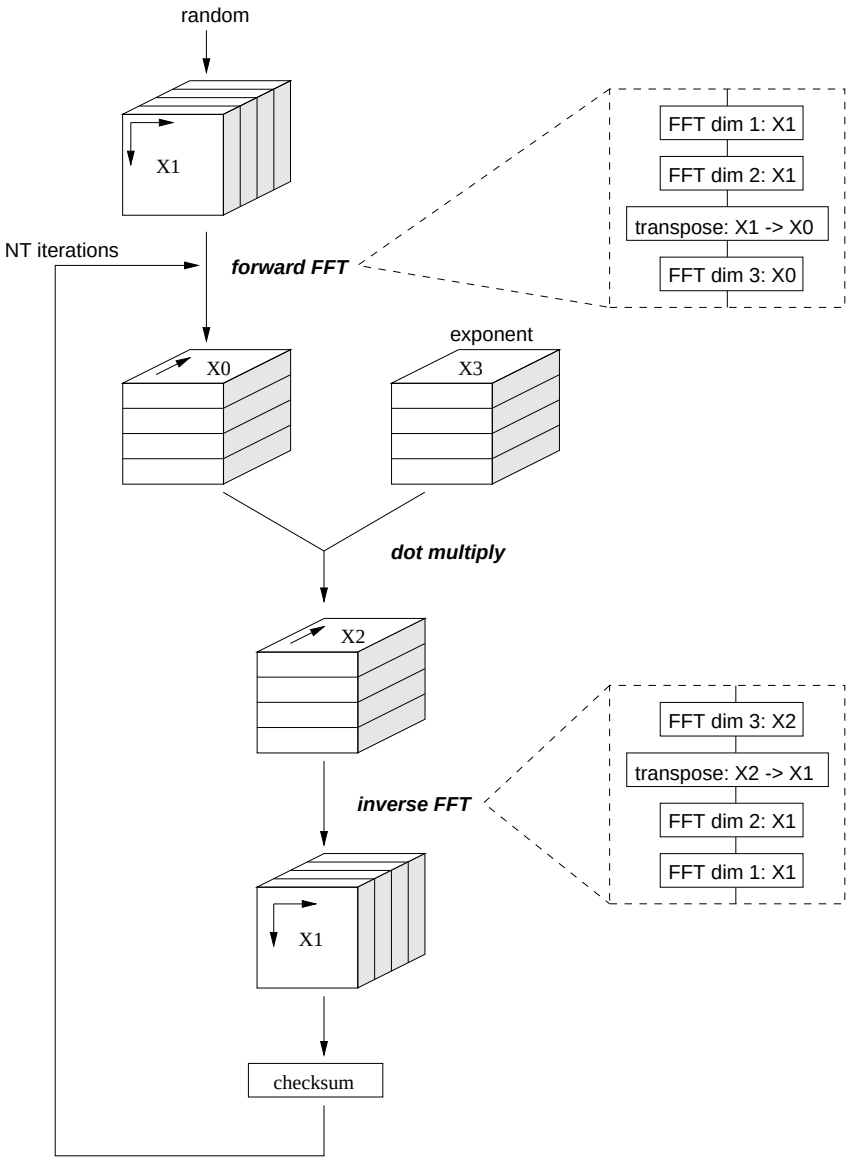


Figure 5.3: Illustrations of FT implementation.

Note: (1) The 3-D arrays X0, X2 and X3 are partitioned along dimension 2; X1 is partitioned along dimension 3; (2) The arrows in the arrays show the direction of the 1-D FFT.

2. The communication for the transpose step is replaced with a global assignment statement.

3. A scratch array that is recast into arrays of different ranks and sizes between subroutines is replaced with multiple arrays of constant rank and size. Although passing an array section in a formal argument is legitimate in HPF, some HPF compilers have difficulty managing array sections.

The HPF F90/Forall version requires the additional step of rewriting all data parallel loops in F90 syntax.

The ZPL implementation allocates the 3-D arrays as *regions* of 2-D arrays; the transpose operation is realized with the ZPL *permute* operator.

5.3 Parallel Performance

In this section we examine the performance of the programs. Because the execution time may be excessive depending on the success of the compilers, we first examine the small problem size (class S), then the programs with a reasonable performance and speedup with the large problem size (class A). Figure 5.4, 5.5 and 5.6 show the aggregate timing for all versions (MPI, HPF, ZPL) and for the small and large problem size (class S, class A). The following discussion will focus on (1) the scalability and (2) the scalar performance, and examine the causes for any problems with (1) and (2).

5.3.1 NAS EP benchmark

In Figure 5.4(a), the first surprising observation is that the IBM and PGI compilers achieve no speedup with the HPF DO loop version although the APR compiler produces a program that scales well (recall that the EP DO loop version is from the APR suite). Inspecting the code reveals that no distribution directives were specified for the arrays, resulting in a default data distribution. Although the default distribution is implementation dependent, the conventional choice is to replicate the array. The IBM and PGI

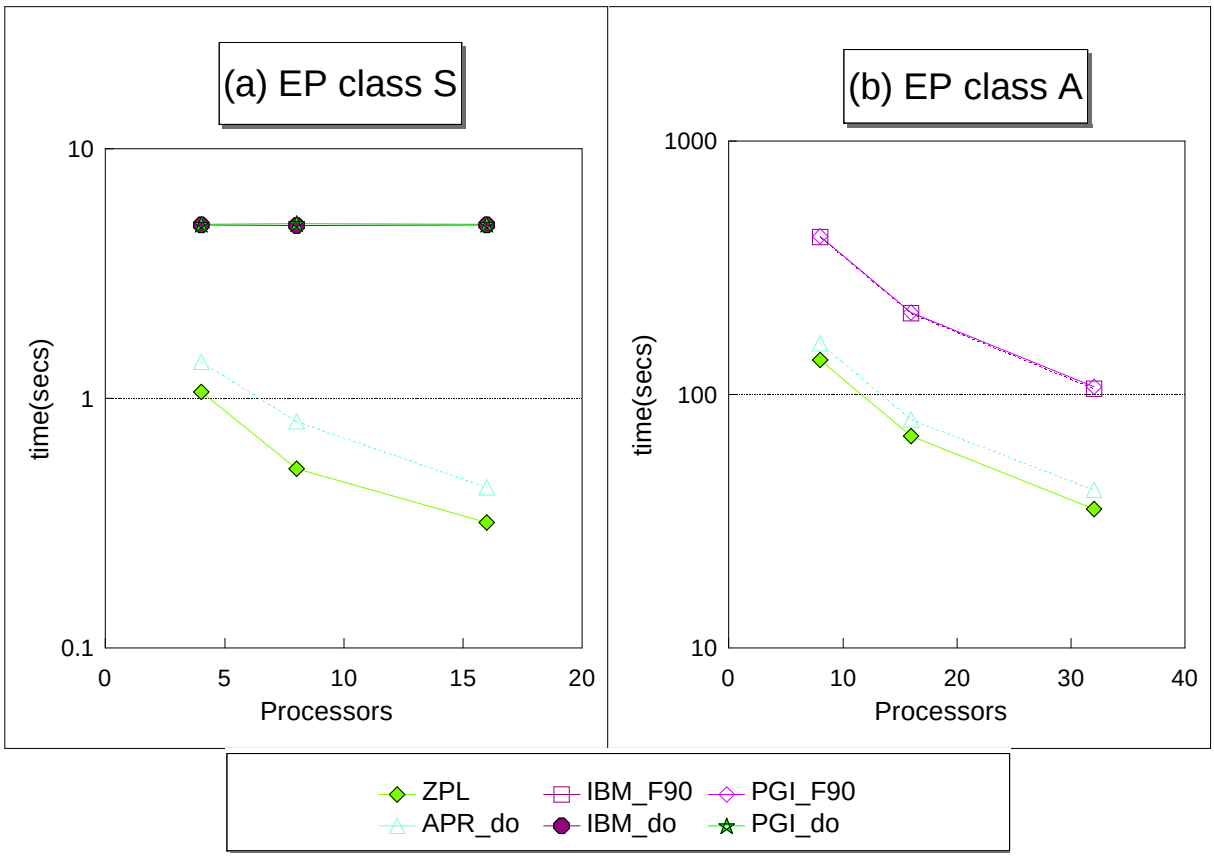


Figure 5.4: Performance for EP (log scale).

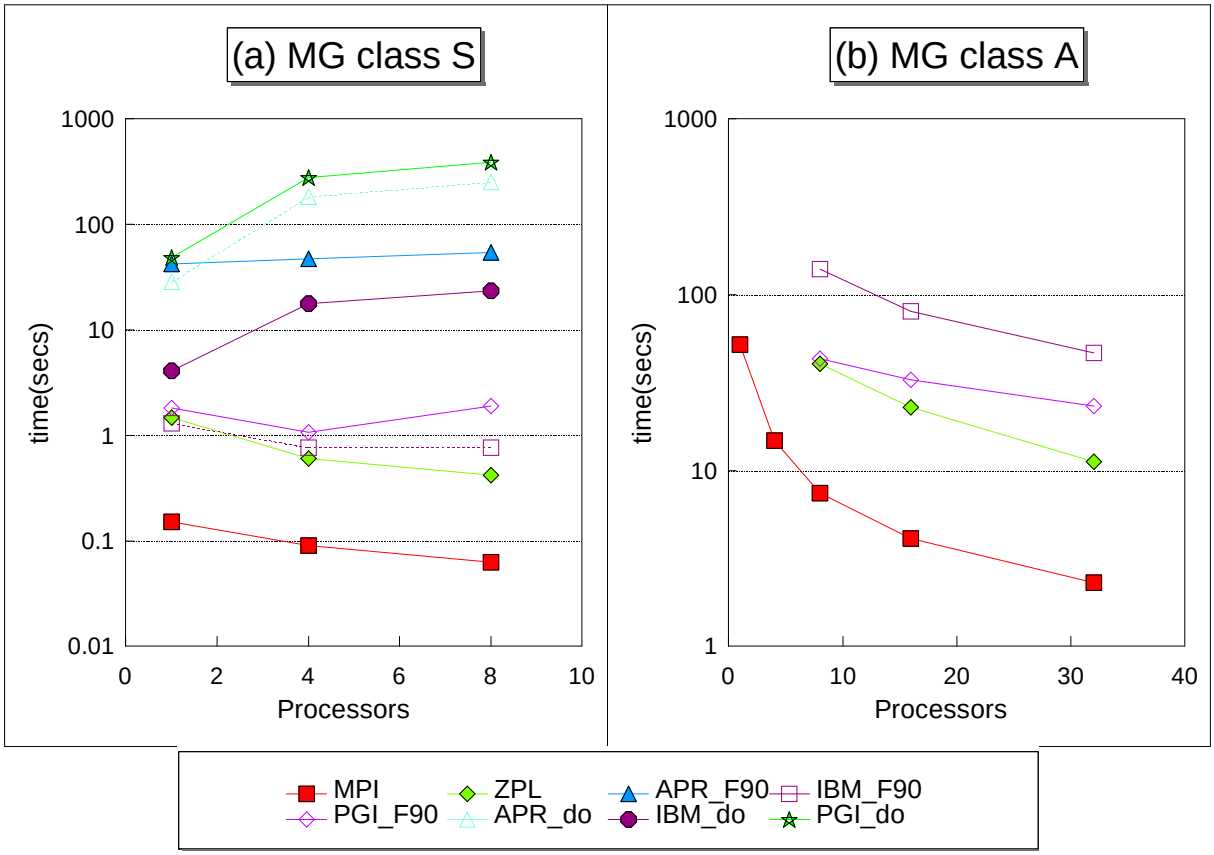


Figure 5.5: Performance for MG (log scale).

Note: APR is not shown in (b) because the execution timed out.

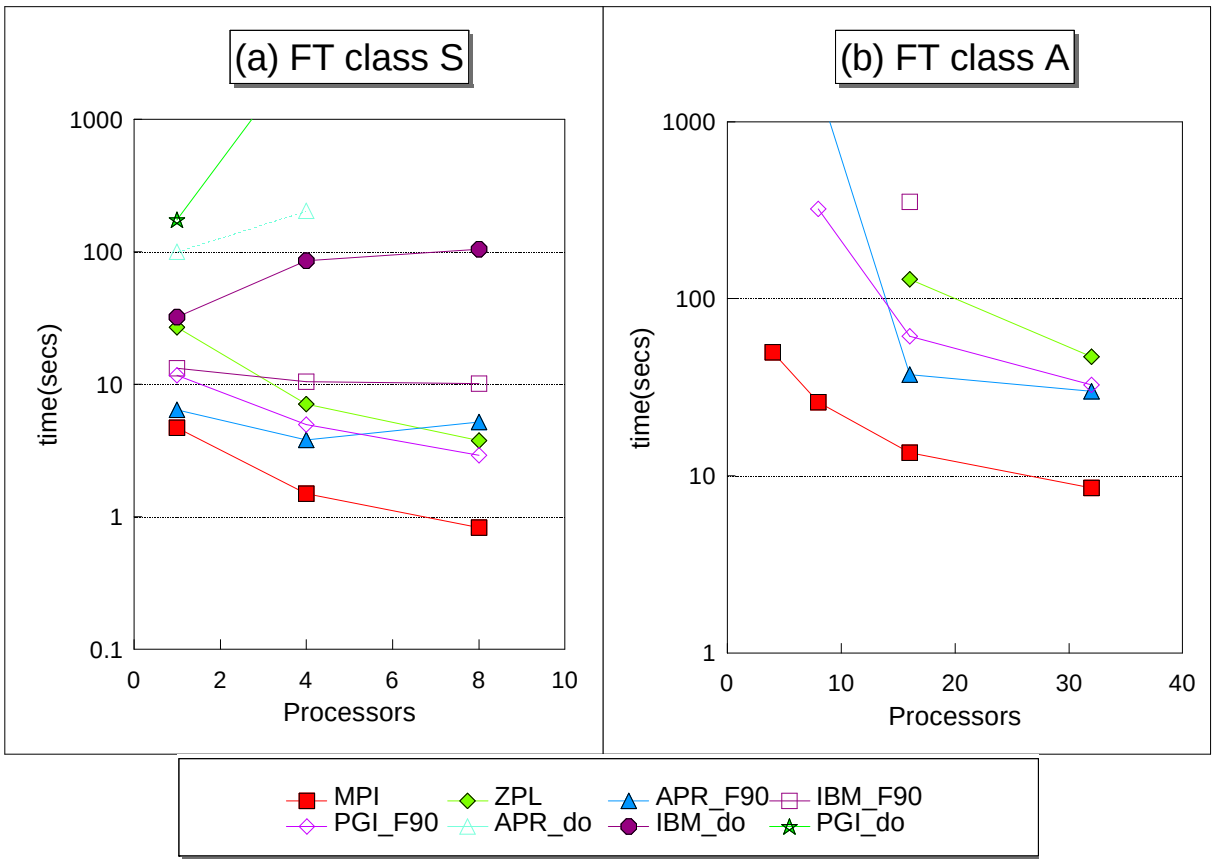


Figure 5.6: Performance for FT(log scale).

compilers distribute the computation by the owner-computes rule[4]; therefore, in order for the program to be parallelized, some data structures must be distributed. Since the arrays in EP are replicated by default, no computation is partitioned among the processors: each processor executes the full program and achieves no speedup.

By contrast, the APR parallelization strategy does not strictly adhere to the owner-computes rule. This allows the main loop to be partitioned despite the fact that none of the arrays within the loop are distributed. Note that the HPF language specification does not specify the default distribution for the data nor the partitioning scheme for the computation. The omission was likely intended to maximize the opportunity for the compiler to optimize; however the observation for EP suggests that the different schemes adopted by the compilers may result in a portability problem with HPF programs.

When directives were inserted to distribute the arrays, it was found that the main array in EP is intended to hold pseudo-random values generated sequentially, therefore there exists a true dependence in the loop computing the values. If the array is distributed, the compiler will adjust the loop bounds to the local partition, but the computation will be serialized.

The HPF F90/Forall version corrects this problem by explicitly distributing the arrays and the IBM and PGI compilers were able to parallelize. The class A performance in Figure 5.4(b) shows that all compilers achieve the expected linear speedup. However, expanding the arrays to express the computation into a more data parallel form introduces overhead and degrades the scalar performance. It is possible for advanced compiler optimizations such as loop fusion and array contraction to remove this overhead, but these optimizations were either not available or not successful in this case.

The ZPL version scales linearly as expected and the scalar performance is slightly better than the APR version despite the C/Fortran difference described earlier.

[4]PGI can also deviate from the owner-computes rule in some case.

5.3.2 NAS MG benchmark

Compared to EP, MG allows a more complete and rigorous test of the languages and compilers. We first discuss the performance for the class S in Figure 5.5(a).

The p=1 column shows considerable variation in the scalar performance with all versions showing overhead of 1 to 2 orders of magnitude over the Fortran/MPI performance.

For the base cases, both the original MPI program and the ZPL version scale well. The ZPL compiler partitions the problem in a straightforward manner according to the region and strided region semantics, and the communication is vectorized with little effort. The scalar performance does however show over a 6x overhead compared to the MPI version.

The HPF DO loop version clearly does not scale with any HPF compiler.

The PGI compiler performs poorly in vectorizing the communication when the computation is expressed with DO loops: the communication call tends to remain in the inner most loop, resulting in a very large number of small messages being generated. In addition, the program uses guards within the loop instead of adjusting the loop bound.

The APR compiler only supports a 1-D processor grid, therefore the 3-D distribution specified in the HPF directives is collapsed by default to a 1-D distribution. This limitation affects the asymptotic speedup but does not necessarily limit the parallelization of the 27-point stencil computation. For one subroutine, the compiler detects through interprocedural analysis an alias between two formal arguments, which constitutes an inhibitor for the loop parallelization within the subroutine. However, the analysis did not go further to detect from the index expressions of the array references that no dependence actually exists. For most of the major loops in the program, the APR compiler correctly partitions the computation along the distributed array dimension, but generates very conservative communication before the loop to obtain the latest value for the RHS and after the loop to update the LHS. As a result, the performance degrades with the number of processors.

The IBM compiler does not parallelize because it detects an output dependence on a

number of variables although the arrays are replicated. In this case, the compiler appears to be overly conservative in maintaining the consistency of the replicated variables. Other loops do not parallelize because they contain an IF statement.

The INDEPENDENT directive is treated differently by the compilers. The PGI compiler interprets the directive literally and parallelizes the loop as directed, while the IBM compiler nevertheless performs a more rigorous dependence check and elects not to parallelize the loop because of detected dependences.

For the HPF F90/Forall version, the IBM and PGI compilers are more successful. The IBM compiler performance and scalability approach ZPL's, while the PGI compiler now experiences little problem in vectorizing the communication. Indeed, PGI's scalar performance now exceeds IBM's. The APR compiler does not result in slowdown but does not achieve any speedup either. It partitions the computation in the F90/Forall version similarly to the DO loop version, but is able to reduce the amount of communication. It continues to be limited by its 1-D distribution as well as an alias problem with one subroutine. Note that the version of MG from the APR suite employs APR's directives to suppress unnecessary communication. These directives are not used in our study because they are not a part of HPF, but it is worth noting that it is possible to use APR's tools to analyze the program and manually insert APR's directives to improve the speedup with the APR compiler.

Given that the DO loop version fails to scale with any compiler, one may conjecture whether the program may be written differently to aid the compilers. The specific causes for the failure of each compiler described above suggest that the APR compiler would be more successful if APR's directives are used, that the PGI compiler may benefit from the HPF INDEPENDENT directive, and that the IBM compiler would require actual removal of some data dependences. Therefore, it does not appear that any single solution is portable across the compilers.

Since the HPF DO version does not scale, the class A data only includes MPI, ZPL and HPF F90/Forall (Figure 5.5(b)). MPI and ZPL again exhibit good speedup but the

ZPL overhead persists. IBM and PGI also achieve speedup but PGI appears to level off quickly while IBM shows yet higher overhead than ZPL. APR on the other hand does not achieve any speedup as predicted with the small problem size; the APR plot is not shown because the execution runs timed out. Note that the MG performance published by APR is competitive with ZPL's performance. However, the MG benchmark made available by APR not only relies on APR's directives but can be compiled by neither IBM nor PGI because of the use of *SEQUENCE*, an incompatible memory layout feature.

MG thus illustrates that (1) HPF programs can achieve some speedup, and (2) when tuned to a specific compiler such as APR, the programs can achieve the same level of performance as ZPL. However, it is difficult to guarantee scalability and portability in HPF programs.

5.3.3 NAS FT benchmark

FT presents a different challenge to the HPF compilers. In terms of the reference pattern, FT consists of a dot product and the FFT butterfly pattern. The former requires no communication and is readily parallelized by all compilers. For the latter, the index expression is far too complex for a compiler to optimize the communication. Fortunately, the index variable is limited to one dimension at a time; therefore the task for the compiler is to partition the computation along the appropriate dimensions. The intended data distribution is 1-D and is thus within the capability of the APR compiler.

Figure 5.6(a) shows the full set of performance results for the small problem size. As with MG, the MPI and ZPL versions scale well and the scalar performance of all HPF and ZPL implementations shows an overhead of 1 to 2 orders of magnitude over the MPI implementation.

For the HPF DO loop version, the APR compiler exhibits the same problem as with MG: it generates very conservative communication before and after many loops. In addition, the APR compiler does not choose the correct loop to parallelize. The discrepancy arises because APR's strategy is to choose the partitioning based on the

array references within the loop. In the program, the main computation and thus the array references are packaged in a subroutine called from the loop so that when the loop is parallelized, the subroutine will operate on the local data. When the APR compiler proceeds to analyze the loops in this subroutine (1-D FFT), it finds that the loops are not parallelizable.

The PGI compiler also generates poor communication, although its principal limitation is in vectorizing the messages. The IBM compiler does not parallelize because of assignments to replicated variables.

The HPF F90/Forall version requires considerable experimentation and code restructuring to arrive at a version that is accepted by all compilers, partly because of differences in supported features among the compilers and partly because of the nested subroutines structure of the original program. All HPF compilers achieve speedup to varying degrees. APR is particularly successful since the principal parallel loop has been moved to the innermost subroutine. Its scalar performance approaches the MPI's performance, although communication overhead limits the speedup. PGI shows good speedup while IBM's speedup is more limited.

For the class A problem size, the memory requirement proves to be considerable since several programs fail the p=8 configuration. Unfortunately, the Cornell system only has 48 wide nodes out of its 512 nodes; this limits the set of the data points. The memory requirement also manifests itself in some superlinear speedup for all HPF and ZPL programs. Nevertheless, the overall observation is that the programs achieve the expected speedup. ZPL scalar performance is lower than the HPF performance; however, when we take into account the 40% slowdown from C to Fortran measured earlier for FT, ZPL parallel performance is comparable to HPF.

5.3.4 Communication

Table 5.2 shows the total number of MPI message passing calls generated and the differences in the communication schemes employed by each compiler. The APR and PGI

Table 5.2: Communication statistics for EP class A, MG class S and FT class S: p=8

Benchmark	version	point-to-point	collective	type of MPI calls
EP (class A)	ZPL	0	120	Allreduce, Barrier
	APR DO loop	31	0	Send, Recv
	IBM F90	70	120	Send, Recv, Bcast
	PGI F90	240	0	Send, Recv
MG (class S)	MPI	2736	40	Send, Irecv, Allreduce, Barrier
	ZPL	9504	56	Isend, Recv, Barrier
	APR F90	126775	8	Send, Recv, Barrier
	IBM F90	9636	32	Send, Recv, Irecv, Bcast
	PGI F90	22191	0	Send, Recv
FT (class S)	MPI	0	104	Alltoall, Reduce
	ZPL	1064	32	Isend, Recv, Barrier
	APR F90	58877	8	Send, Recv, Barrier
	IBM F90	728	258048	Send, Irecv, Bcast
	PGI F90	64603	0	Send, Recv

compilers only use the generic send and receive while the IBM compiler also uses the nonblocking calls and the collective communication; this may have ramifications in the portability of the IBM compiler to other platforms. The ZPL compiler uses nonblocking MPI calls to overlap computation with communication as well as MPI collective communication.

5.3.5 Data Dependences

HPF compilers derive parallelism from the data distribution and the loops that operate on the data. Loops with no dependences are readily parallelized by adjusting the loop

bounds to the local bounds. Loops with dependences may still be parallelizable but will require analysis; for instance, the IBM compiler can detect some dependence patterns of loops that perform a reduction and generate the appropriate HPF reduction intrinsic. In other instances, loop distribution may isolate the portion containing the dependence to allow the remainder of the original loop to be parallelized. To approximately quantify the degree of difficulty that a program presents to the parallelizing compiler in terms of dependence analysis, we use the following simple metric:

$$\frac{count\ of\ all\ loops\ with\ dependences}{count\ of\ all\ loops}$$

A value of 0 would indicate that all loops can be trivially parallelized, while a value of 1 would indicate that whether any loop is parallelizable depends on the analysis capability of the compiler. Using the KAPF tool, we collect the loop statistics from the benchmarks for the major subroutines; they are listed in Table 5.3. This metric is not complete since it does not account for the data distribution; for instance, for 3 nested loops and a 1-D distribution, only 1 loop needs to be partitioned to parallelize the program and 2 loops may contain dependences with no ill effect. However, the metric gives a coarse indication for the demands on the compiler.

The loop dependence statistics show clear trends that correlate directly with the performance data. We observe the expected reduction in dependences from the DO loop version to the F90/Forall version. The reduction greatly aids the compilers in parallelizing the F90/Forall programs, but also highlights the difficulty with parallelizing programs with DO loops.

For MG, the difference is significant; the array syntax eliminates the dependences in most cases. Some HPF compilers implement optimizations for array references that are affine functions of the DO loop indices, particularly for functions with constants. These optimizations should have been sufficient for the MG DO loop version, however it does not appear that they were successful. Note that the loops in the subroutine *norm2u3* are replaced altogether with the HPF reduction intrinsics.

For FT, the low number of dependences in *fftpde* comes from the dot-products which

Table 5.3: Dependence ratio $\frac{m}{n}$ for EP, MG and FT.

Note: m is the count of loops with data dependences or subroutine calls, and n is the total loop count.

	subroutine	DO	F90/Forall
EP	embar	3/5	1/31
	vrandlc	1/1	-
	get_start_seed	-	1/1
FT	fftpde	2/16	2/16
	cfft3	0/6	0/6
	cffts1	2/6	1/7
	cffts2	2/6	1/7
	cfftz	3/5	1/4
	fftz2	3/3	3/4

	subroutine	DO	F90/Forall
MG	hmg	1/2	1/27
	mg3P_up	1/1	1/22
	mg3P_down	1/1	1/1
	psinv	4/4	0/6
	resid	4/4	0/6
	rprj3	4/4	0/6
	interp	7/21	0/30
	norm2u3	3/3	0/0
	comm3	0/6	0/18

are easily parallelized. The top-down order of the subroutines listed also represents the nesting level of the subroutines. The increasing dependences in the inner subroutine reflect the need to achieve parallelism at the higher level. As explained earlier, this proves to be a challenge to the APR compiler which focuses on analyzing individual loops to partition the work.

5.4 Conclusion

Our objective in this chapter has been to subject the current state of the art compilers for data parallel languages to more substantive applications. Three NAS benchmarks were studied across three current HPF compilers and a ZPL compiler. We examined different styles of expressing the computation in HPF and we also consider the same benchmarks written in MPI to understand the limits of the performance.

The three HPF compilers show a general difficulty in detecting parallelism from DO loops. They are more successful with the F90 array syntax and the Forall construct, although even in this case the success in parallelization is not uniform. Significant variation in the scalar performance also exists between the compilers.

The ZPL compiler shows consistent speedup and performance competitive with the HPF compilers.

While differences between compilers will always be present, the differences must preserve a certain performance model in order for program portability to be maintained in the language. In other words, the user must be able to use any compiler to develop a program that scales, then have the option of migrating to a particular machine or compiler for better scalar performance. This requires a tight coupling between the language specification and the compiler in the sense that the compiler must reliably implement the abstraction provided in the language. To this end, the language specification must serve as a consistent *contract* with the programmer, or more formally, the language must provide a concise performance model.

In the case of HPF, the results point to two difficulties.

First, while the HPF directives and constructs provide information on the data and computation partitioning, the sequential semantics of Fortran leave many potential dependences in the program. An HPF compiler must analyze these dependences, and when unable to do so, it must make a conservative assumption. Although this analysis capability differentiates different vendor implementations, the difficulty for the compilers to parallelize reliably leads to a difficulty for the user in predicting the parallel behavior and thus the speedup of the program. A direct result is that the user needs to continually experiment with the compilers to learn their actual behavior. In doing so, the user is effectively supplementing the performance model provided by the language with empirical information. Yet, such an enhanced model tends to be platform specific and not portable.

Second, the optional nature of the directives, while fostering compatibility and a

smoother transition from a current language, leads to an uncertain performance model for the user. In other words, it is not clear how much effort from the user is necessary or sufficient; for instance, the INDEPENDENT directive may or may not parallelize a loop depending on the compiler implementation.

In this respect, the ZPL language addresses some of these problems by providing a clear demarcation between parallel and sequential execution. The results demonstrate that ZPL offers a consistent performance model and scalable performance.

The results also show that significant overhead remains in all implementations compared to the MPI programs. One source for the overhead is the large number of temporary arrays generated by the compiler across subroutine calls and parallelized loops. They require dynamic allocation/deallocation and copying, and generally degrade the cache performance. It is clear that to become a viable alternative to explicit message passing, compilers for data parallel language must achieve a much lower overhead.

Chapter 6

Mighty Scan, parallelizing sequential computation

6.1 Introduction

The general objective for a parallelizing compiler is to analyze and detect the dependences in the program so that when there are none the computation can be scheduled to proceed in parallel. However, when a true dependence exists, serialization occurs and a different approach must be employed to achieve parallelism. In this discussion, we focus on the case where the dependence occurs along one dimension of one or several arrays. Such a computation typically involves traversing the array(s) and updating each element using the values of the preceding elements. Because a true dependence exists, the computation is conceptually sequential. A simple example is the parallel prefix operation (scan) on an array. A more complex example involving multiple arrays and arithmetic operations is the forward elimination and backward substitution steps in a solver for linear equations.

The solution for this problem exists in many forms depending on the particular case. When the operator is commutative and associative and only one array is involved, the computation is known as the parallel prefix operation. Because it is frequently used and an efficient parallel algorithm exists, parallel prefix is often supported directly

in communication interfaces such as MPI (MPI_Prefix) and in high level data parallel languages such as HPF and ZPL. Since these implementations are well optimized, we can consider the parallel prefix problem to be solved.

However, when the operator is not commutative or associative and multiple arrays are involved, no direct support currently exists in either the libraries or the languages. For message passing programs, this does not present a serious obstacle since the relatively low programming level allows considerable freedom in implementing any algorithm. In this case, pipelining has proven to be very effective when additional coarse grain parallelism exists in addition to the sequential computation. When the program is written in a data parallel language, the solution is also straightforward if the programmer has freedom in choosing the array partitioning scheme. For instance, if the sequential computation proceeds along the first dimension of a 2-D array (as illustrated in Figure 6.1(a)), the array can simply be partitioned along the second dimension onto a 1-D processor grid so that there is no interprocessor dependence. This allows each processor to proceed independently. However, it is often the case that the best partitioning scheme for the overall problem requires the first dimension to be partitioned also, for instance a 2-D array onto a 2-D processor grid as shown in Figure 6.1(b). In this case, all processors in the first dimension will be serialized during the sequential computation.

The problem described above thus sets the stage for our discussion. Assume: (1) a program written in a data parallel language (HPF or ZPL), (2) a computation that is semantically sequential along dimension i of some array(s) and, (3) a partitioning scheme that requires dimension i to be distributed onto a processor grid. The goal is to avoid the serialization of the processor set onto which the sequential computation is mapped.

In the remainder of this chapter, Section 6.2 will describe a number of solutions, their implementations in HPF and ZPL and their shortcomings. Then Section 6.4 will propose the specifications for a new language construct that offers the best solution.

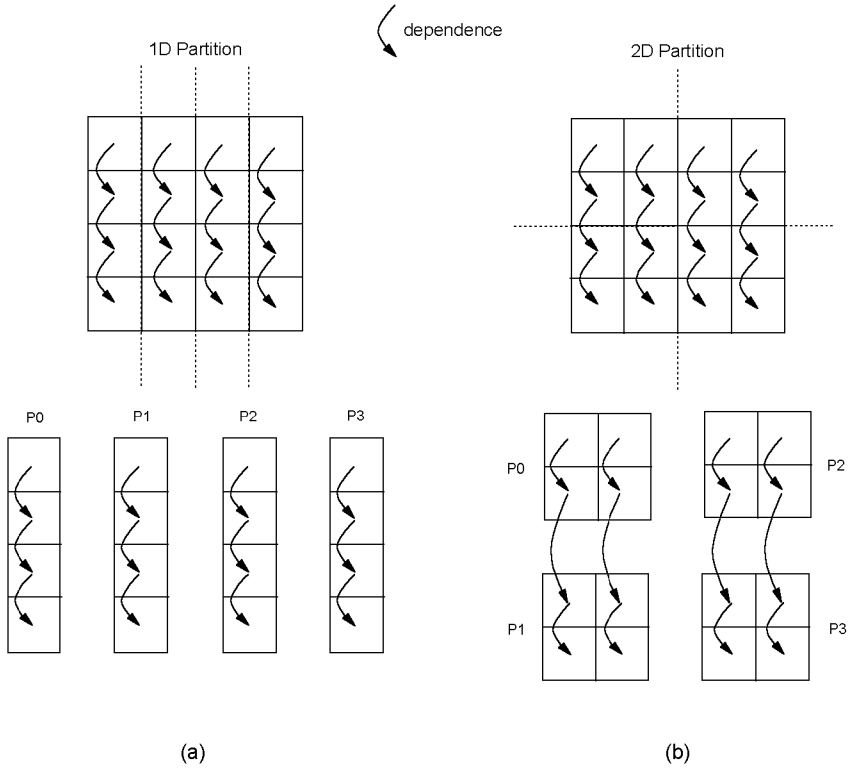


Figure 6.1: Two methods for array partitioning.

Note: (a) no dependence across processors, fully parallel; (b) dependence exists, some processors serialized.

6.2 A case study

For the case study, we will use the widely studied *tomcatv* benchmark since it embodies the essential characteristics while remaining sufficiently simple to facilitate our analysis. A much more complex and realistic case is the NAS SP and BT benchmarks, which are large scale partial differential equation (PDE) solvers, typically used in computational fluid dynamics (CFD). The version of tomcatv under study is from the suite of HPF benchmarks published by APR, which has been restructured and annotated with directives for HPF. The computation consists of a 2-phase iteration over several 2-D arrays:

Phase 1 is a 9-point stencil computation and Phase 2 is a solver for a tridiagonal system of linear equations. Phase 2 exhibits a true dependence along the first dimension; therefore it motivates distributing the array along the second dimension. However, Phase 1 favors a 2-D partition for better scalability.

The forward elimination step of Phase 2 consists of the following (the backward substitution step is similar):

```
DO i = 2 , n
   DO j = 2 , m
      r(1,i) = aa(j,i) * d(j - 1,i)
      d(j,i) = 1. / (dd(j,i) - aa(j - 1,i) * r(1,i))
      rx(j,i) = rx(j,i) - rx(j-1,i) * r(1,i)
      ry(j,i) = ry(j,i) - ry(j-1,i) * r(1,i)
   ENDDO
ENDDO
```

The characteristics of this computation can be summarized as:

1. Sequential dependence in the j dimension: no parallelism.

2. No dependence in the i dimension: available parallelism.

3. Multiple arrays and arithmetic operations are involved.

Since tomcatv is a small program, the difference in the partitioning choice may not be significant[1]. However for larger problem sizes or larger programs such as the NAS SP and BT benchmark, the asymptotic difference becomes clear. Specifically, Naik showed that for SP on the IBM SP/1, a 3-D partitioning can be 66% faster than a 1-D partitioning on 16 processors [Naik 94].

6.2.1 Idealized execution

Figure 6.2 shows three possible parallel executions of the loops, assuming that the $m \times n = 4 \times 4$ array is distributed onto a $P_r \times P_c = 2 \times 2$ processor grid. The solid time line for each

[1] This proves to be a mitigating circumstance for the APR compiler which only accepts 1-D processor grid.

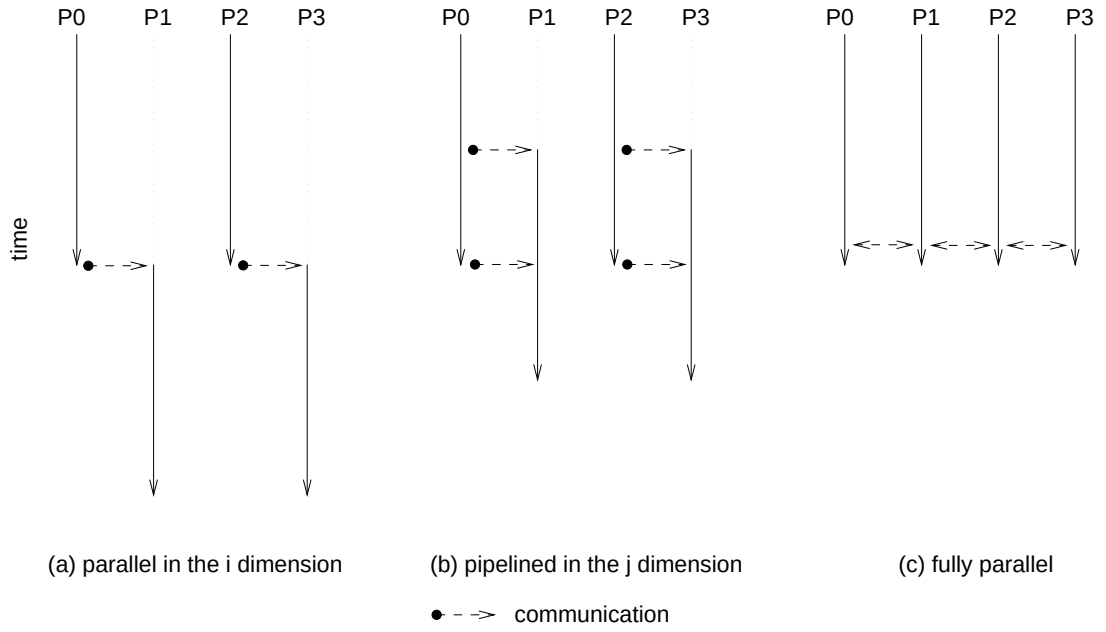


Figure 6.2: Three possible parallel executions.

processor represents the computation required for the 2×2 array section that it owns. In all cases, loop i is fully parallel; the only difference is in how loop j is parallelized.

In case (a), loop j remains serialized: the P_r processors fail to execute in parallel and the speedup is limited to P_c. In case (b), loop j still executes sequentially, but by employing pipelining, partial results are forwarded to allow the waiting processors to initiate their execution earlier. The overhead to start and finish the pipeline will degrade the overall speedup to bP_cP_r, where $b < 1$. In case (c), loop j proceeds effectively in parallel. This is possible if an efficient parallel algorithm exists for the j loop: if the parallelization overhead can be limited to $O(\log P_r)$, a speedup of $O(\frac{P_cP_r}{\log P_r})$ can be achieved.

6.2.2 An algorithmic approach

We now consider an algorithmic approach for parallelizing in the j dimension. It is algorithmic since it relies on unique properties of the intended computation instead of

the language construct. This approach parallelizes the scan operation itself, resulting in an execution similar to Figure 6.2(c). One important advantage is that no other source of parallelism is required (e.g., parallelism in the i dimension).

Parallel prefix is an example of a sequential computation that can be parallelized effectively with an overhead of $O(\log P)$. The parallel algorithm in this case requires that the operator be associative so that the partial results can be combined arbitrarily. This requirement is too restrictive for the general scan operation that we are considering. However, with careful manipulation, it may be possible to factor out from the sequential computation certain components that can be computed in parallel. Then we can examine if the remaining components can be computed with a reasonable overhead. These two steps would constitute the two phases of a parallel implementation of scan: (1) compute locally and (2) communicate and update.

Consider a scan computation that is slightly more complex than a simple summation and that is similar (but not identical) to that found in tomcatv. For array $x(1:16)$ and $b(1:16)$:

```
DO i=2,16
   x(i) = x(i) + x(i-1) * b(i-1)
ENDDO
```

Although the code has the structure of a scan operation, we cannot use the parallel prefix algorithm because the combination of + and * is not associative. However, if the terms are expanded completely, the computation can be expressed as follows:

$$
\begin{aligned}
x_1 &= x_1 \\
x_2 &= x_1 b_1 + x_2 \\
x_3 &= x_1 b_1 b_2 + x_2 b_2 + x_3 \\
x_4 &= x_1 b_1 b_2 b_3 + x_2 b_2 b_3 + x_3 b_3 + x_4 \\
&\cdots \\
x_k &= \sum_{j=1}^{k-1} (x_j \prod_{i=j}^{k-1} b_i) + x_k
\end{aligned}
$$

Let processor P own a section $(m : n)$ of the arrays. Then in computing each new element in its array section, the summation expression can be split into two components. For element x_n specifically:

$$x_n = \sum_{j=1}^{m-1}(x_j \prod_{i=j}^{n-1} b_i) + \sum_{j=m}^{n-1}(x_j \prod_{i=j}^{n-1} b_i) + x_n$$

Note that the second term involves only local elements in the array section $(m : n)$; therefore processor P can proceed to compute this component independently. Note also that for the first term, the upper limit for the summation is $m-1$ and for the product is $n-1$. This allows us to factor the common b, thus further splitting the first term into three factors:

$$x_n = \prod_{j=m}^{n-1} b_j * b_{m-1} * (\sum_{j=1}^{m-1}(x_j \prod_{i=j}^{n-1} b_i) + x_{m-1}) + \sum_{j=m}^{n-1}(x_j \prod_{i=j}^{n-1} b_i) + x_n$$

Clearly for the first term, the first factor $\prod_{j=m}^{n-1} b_j$ can be computed locally. The third factor is simply the value for x_{m-1} computed by its owner processor, which incidentally also owns the second factor b_{m-1}.

A parallel algorithm can finally be described:

1. Compute locally: $x_k = \sum_{j=m}^{k-1}(x_j \prod_{i=j}^{k-1} b_i) + x_k, m \leq k \leq n$
2. Compute locally: $\prod_{j=m}^{n-1} b_j$
3. Receive from preceding processor: $x_{m-1} * b_{m-1}$
4. Update x_n first using the results from 1, 2 and 3
5. Send to next processor: $x_n * b_n$
6. Update remaining elements $x_{m:n-1}$

Considering the complexity, the local computation in steps 1, 2 and 6 requires $O(N/P)$ where N is the array size and P is the number of processors. Although the communication is sequential, it occurs after the main parallel computation and only one operation is interposed between each message; therefore the serialization effect is limited

to $O(P)$. A compiler that implements this algorithm may generate the following SPMD program.

```
/* local computation */
for (i=mylo+1; i<=myhi; i++) {
    x(i) = x(i) + x(i-1) * B(i-1);
    myB = myB * B(i);
}
/* update and forward */
recv_val = receive(ProcID-1);
send_val = (myB * recv_val + x(myhi)) * B(myhi);
send(ProcID+1, send_val);
/* local update */
for (i=mylo; i<=myhi; i++) {
    x(i) = x(i) + recv_val;
    recv_val = recv_val * b(i);
}
```

The exercise above shows that it is possible to parallelize a scan operation that is not associative. However it also demonstrates that the parallelization for a general scan requires significant analysis that is not easily automated by a compiler nor conveniently performed by the user. This motivates us to consider simpler alternatives.

6.2.3 Pipelining

Pipelining is a ubiquitous and very effective technique for parallelism. One of its advantages is simplicity: the concept is easily understood and the technique can be easily applied. The regular nature of pipelining also lends itself well to performance tuning to balance the trade-off between platform specific parameters. Pipelining can occur at virtually any level of granularity. Microprocessors employ pipelining effectively at the instruction level to increase the throughput. We are interested in pipelining at a coarser level where an algorithm is implemented. In this case, a prerequisite for pipelining is the availability of additional parallelism outside the computation having the true dependence.

Figure 6.2(b) illustrates the execution for tomcatv if the loop is pipelined in the j dimension. The goal is to make available as early as possible any result for which a dependence exists. Thus instead of completing all j iterations before performing any communication, processor $P0$ and $P2$ would send the new values at the end of each j iteration to processor $P1$ and $P3$ so that they can initiate their execution immediately.

Currently, pipelined execution is not provided as a programming construct in any data parallel language. Since a compiler is free to perform the analysis and generate any execution schedule, one may question whether a compiler can generate the pipelined code from the existing data parallel constructs? Although no fundamental barrier exists, there are several difficulties.

First, the compiler must be able to detect the opportunity for pipelining. The conventional DO loop has been shown to be difficult to analyze in general. At the same time, the F90 and Forall semantics in HPF and the region semantics in ZPL call for a synchronization after each array statement, while pipelining typically spans across multiple statements. In each case, the dependences are over-specified and the compiler must perform analysis to detect the opportunity.

Second, the optimization objective in pipelining may be in conflict with other optimization techniques[Choi & Snyder 97]. For instance, message vectorization attempts to combine multiple messages into one to amortize the message startup cost[Choi & Snyder 97]. When applied to the j loop in tomcatv, this optimization will favor moving the communication outside the j loop so that all individual messages in the loop will be combined into one message. This transformation is productive in many cases, but in this case it serializes the execution (see Figure 6.2). At the same time, applying pipelining aggressively may not be beneficial since attempting to make newly computed values available to waiting processors as early as possible may easily increase the number of messages while decreasing the message size. In this case, the high message startup cost may dominate the benefit of pipelining.

Third, the pipelining is not a part of the language semantics; therefore, a user who

wishes to implement a pipelined algorithm has no means to directly express it. Rather, the user must rely on the optimization features of the compiler.

6.3 Implementations by HPF and ZPL

Having presented the desired solutions, we now examine how the sequential computation is expressed and implemented in HPF and ZPL. We can expect that the compilers will have little problem generating parallel codes for the i dimension in tomcatv. Although parallelizing the j dimension is difficult, no fundamental barrier prevents the compiler from parallelizing the loop[Cytron 86]. In particular, we are interested in finding if the compilers can detect and implement pipelining as described in the previous section.

Table 6.1 shows the tomcatv forward elimination step expressed in HPF using DO loops and F90 array statements, and in ZPL using dynamic regions. Table 6.2 shows the pseudo-code for the implementations by HPF and ZPL. We will consider the PGI and IBM HPF compiler and the ZPL compiler; APR is not included because it is limited to partitioning one array dimension.

6.3.1 DO loop implementation

For programs using DO loops, HPF provides the *INDEPENDENT* directive to customize the dependences (Figure 6.1(a)). In our case, this approach provides the compiler with the most precise information about the true dependences since an HPF compiler can easily assume that the i loop has no loop dependences while the j loop does. With this information, the IBM and PGI compilers generate the implementations described by the SPMD pseudo-codes in Table 6.2. The principal distinction between the implementations is the placement of the communication and computation, and whether the loops are parallelized.

The DO loop version is observed in two variations to test the compiler analysis: the *INDEPENDENT* i loop is placed as the outer loop in DO(1) and as the inner loop in DO(2). Examining the placement of the communication among the computation in

Table 6.1: The forward elimination step from tomcatv.
Note: The arrays are distributed onto a 2-D processor grid.

(a) HPF DO loop version:

```
!HPF$ INDEPENDENT
DO i = 2 , n
  DO j = 2 , m
    r(1,i) = aa(j,i) * d(j - 1,i)
    d(j,i) = 1. / (dd(j,i) - aa(j - 1,i) * r(1,i))
    rx(j,i) = rx(j,i) - rx(j-1,i) * r(1,i)
    ry(j,i) = ry(j,i) - ry(j-1,i) * r(1,i)
  ENDDO
ENDDO
```

(b) HPF F90 version:

```
DO j = 2 , m
  r(1,2:n) = aa(j,2:n) * d(j - 1,2:n)
  d(j,2:n) = 1. / (dd(j,2:n) - aa(j-1,2:n) * r(1,2:n))
  rx(j,2:n) = rx(j,2:n) - rx(j-1,2:n) * r(1,2:n)
  ry(j,2:n) = ry(j,2:n) - ry(j-1,2:n) * r(1,2:n)
ENDDO
```

(c) ZPL version:

```
for j:= 2 to m do
[[j, 2..n]] begin
    R:=AA*D@north;
    D:=1.0/(DD-AA@north*R);
    Rx:=Rx-Rx@north*R:
    Ry:=Ry-Ry@north*R;
  end;
end
```

Table 6.2, we find that for DO(1), the IBM compiler follows a straightforward approach by serializing the processors along the j dimension. PGI avoids the serialization by redistributing the 4 arrays involved in the computation from 2-D to 1-D so that they are only partitioned along the i dimension. Then each processor can proceed independently along the j loop, and upon exiting the loops, the arrays are redistributed to return to their original distribution. The array redistribution overhead is significant; therefore PGI's approach is not likely to be scalable[2].

The DO(2) variation is functionally equivalent to DO(1). The IBM compiler is able to determine this fact and generate the same code as DO(1), but the PGI compiler generates a different implementation. Instead of redistributing as before, communication is inserted to fetch all RHS before and to update all LHS after the i loop. This scheme

[2]PGI's approach also deviates from the owner-computes rule, illustrating that owner-computes is not uniformly enforced.

Table 6.2: Pseudo-code for the tomcatv segment by HPF and ZPL compilers.
Note: The *INDEPENDENT* loop is the outer loop in DO(1) and the inner loop in DO(2).

version	IBM	PGI	ZPL

DO (1)	sendrev(aa)
	recv(d)
	recv(rx)
	recv(ry)
	do j=part
	  do i=part
	  r, d, rx, ry = ...
	  enddo
	enddo
	send(d)
	send(rx)
	send(ry)
		sendrecv(aa)
		sendrecv(d)
		sendrecv(rx)
		sendrecv(ry)
		do i=part
		  do j=full
		  r, d, rx, ry = ...
		  enddo
		  enddo
		sendrecv(rx)
		sendrecv(ry)
		sendrecv(d)
			N/A

DO (2)	
	sendrev(aa)
	recv(d)
	recv(rx)
	recv(ry)
	do j=part
	  do i=part
	  r, d, rx, ry = ...
	  enddo
	enddo
	send(d)
	send(rx)
	send(ry)
		do j=full
		  sendrecv(aa)
		  sendrecv(d)
		  sendrecv(dd)
		  sendrecv(rx)
		  sendrecv(ry)
		  do i=part
		  r, d, rx, ry = ...
		  enddo
		  sendrecv(rx)
		  sendrecv(ry)
		  sendrecv(d)
		enddo
			N/A

array	replicate(aa)
	do j=full
	
	  sendrecv(d)
	  do i=full
	  r=...
	  enddo
	
	  do i=part
	  d=...
	  enddo
	  sendrecv(rx)
	  do i=part
	  rx=...
	  enddo
	  sendrecv(ry)
	  do i=part
	  ry=...
	  enddo
	enddo
		
		do j=full
		  sendrecv(aa)
		  sendrecv(d)
		  do i=full
		  r=...
		  enddo
		  sendrecv(aa)
		  do i=part
		  d=...
		  enddo
		  sendrecv(rx)
		  do i=part
		  rx=...
		  enddo
		  sendrecv(ry)
		  do i=part
		  ry=...
		  enddo
		enddo
			
			do j=full
			  sendrecv(aa)
			  sendrecv(d)
			  do i=full
			  r=...
			  enddo
			  sendrecv(aa)
			  do i=part
			  d=...
			  enddo
			  sendrecv(rx)
			  do i=part
			  rx=...
			  enddo
			  sendrecv(ry)
			  do i=part
			  ry=...
			  enddo
			enddo

not only serializes the j dimension but also results in more communication.

6.3.2 An array oriented approach: F90 and ZPL

Array semantics allow independent array operations to proceed in parallel. In this case, the available parallelism is in the i dimension. Therefore an array oriented implementation would apply each statement in the loop across the array section that spans the entire i dimension. Table 6.1(b) and (c) show the HPF version using the F90 array syntax and the ZPL version. The SPMD implementation by the compilers are shown in Table 6.2.

Note that although these two versions are similar in the information provided to the compiler, there are subtle differences originating from the language designs. Specifically, the 2-D array in ZPL is declared as a 2-D region to allow a 2-D partitioning, but because a region imposes no execution order, the sequential dependence in the j dimension must be enforced through a dynamic region, which may incur more runtime overhead than an HPF implementation.

The implementations are remarkably similar across compilers and languages thanks to the implied synchronization after each array statement. The communication required for each statement is generated separately and the loops that implement each statement are placed in the same order as the program statement. The only slight variation is that the IBM compiler elects to vectorize the messages for array aa and move its communication out of the j loop. This consistency contributes to the programming model in making it more predictable. However, for the sequential computation being expressed, we find that the array approach strongly enforces a serialization of the j dimension.

In summary, neither HPF nor ZPL provides direct support for pipelining sequential computation. When we attempt to express the computation in a form that may lead to pipelined execution, we found that the array semantics are too restrictive, while the DO loop annotated with *INDEPENDENT* results in either a serialized execution or an implementation that involves too much communication.

6.4 A new construct for ZPL

Our investigation thus far has shown no satisfactory solution for the problem being considered. While data parallel applications typically contain an abundance of parallelism, some data dependences will always exist that require special consideration. In this respect, we have shown that an algorithmic approach, while possible, does not yield a general solution. On the other hand, the pipelining approach is regularly employed in message passing programs to manage sequential computations. The available parallelism in the data parallel applications easily satisfies the prerequisite for pipelining.

Throughout this thesis we have also demonstrated that correct modeling is critical to the portability, the scalability and the ease of use of a language. With respect to the ease of use, pipelining is a high level abstraction that captures a highly effective programming technique. With respect to scalability, pipelining reduces the serial section of the computation, which otherwise will limit the overall scalability (Ahmdal's Law). To ensure portability, the language behavior must be consistent and predictable, yet the previous sections have shown that relying on the compiler to detect the opportunity to pipeline is unreliable. This motivates direct language support for pipelining that will serve as a contract between the programmer and the compiler.

Following these arguments, we propose the following construct called *Mighty Scan*. The construct is presented within the context of ZPL since the language is relatively free of legacy that may otherwise introduce unnecessary complications.

```
                SCAN i := I_b to I_e DO
[[..., i, ...]] begin
                    statement;
                    statement;
                    ...;
                end;
```

The construct has the following semantics: the block of statements serves as a computational template that is applied along the specified dimension, for the specified index

Table 6.3: tomcatv expressed using SCAN and the resulting SPMD code.

(a) tomcatv using SCAN:

```
                SCAN j:= 2 to m DO
[[j,2..n]]      begin
                   R:=AA*D@north;
                   D:=1.0/(DD-AA@north*R);
                   Rx:=Rx-Rx@north*R:
                   Ry:=Ry-Ry@north*R;
                end;
```

(b) SPMD code:

```
sendrecv(aa)
do j=full/chunk
  recv(aa,d,rx,ry)
  do chunk
    do i=part
      r=...
      d=...
      rx=...
      ry=...
    enddo
  enddo
  send(d,rx,ry)
enddo
```

set and in the specified order. Let R_n be the rank of the region, R_i the scanned dimension, and P_i be the dimension of the processor grid onto which R_i is distributed. If $R_n = 1$, no additional parallelism is available and the $SCAN$ operates as a sequential DO loop and yields no benefit. If $R_n > 1$, the compiler is to set up a pipeline by ensuring that each processor in P_i forwards its partial results in computing an iteration of R_i as early as possible to the next processor. The compiler can optimize further by balancing the tradeoff between the computation granularity and the message frequency.

Note that unlike applying a region to a block of statements, the $SCAN$ semantics do not have the implied synchronization at the end of each statement. The synchronization is replaced instead by the ordering of the scan index.

For the scan operation to be well defined, the construct has the following restrictions:

- For all arrays that are assigned new values in the scan (i.e., they appear on the LHS), their values can be referenced using @ (i.e., they can also appear on the RHS), but only directions in the scan dimension are allowed. This restricts the data dependence to the dimension for which an order will be enforced.

- Scanning multiple dimensions is done by nesting the $SCAN$, with the inner most

SCAN loop being completed first.

Table 6.3 shows the tomcatv example expressed using *SCAN* and the expected SPMD code.

6.5 Conclusions

In this chapter we consider a pattern of computation that occurs frequently, yet for which the current data parallel languages do not provide the facility to parallelize effectively. The computation involves a recurring data dependence; therefore it is conceptually sequential. Although the simple case (parallel prefix) has an effective parallel algorithm that is widely implemented, the general case does not have a general algorithmic solution.

On the other hand, pipelining is regularly and effectively employed in message passing programs to parallelize sequential computation. It is therefore intuitive to apply the same technique for this type of computation in the data parallel program. Although it is possible for the compiler to infer from the existing syntax and implement a pipelined computation, several problems arise, the most important of which is that the functionality is not a part of the performance model. In other words, a user who wishes to express a pipelined algorithm for performance cannot be guaranteed that it will be implemented. Indeed, a case study using the current HPF and ZPL compilers reveals that no compiler recognizes and implements the program tomcatv as a pipeline.

A new construct called Mighty Scan is thus proposed for ZPL that satisfies all the necessary requirements.

1. Ease of use: it implements pipelining as a high level abstraction, hiding the low level detail from the user.

2. Scalability: the technique is well proven in its effectiveness; the user has full control over the computation granularity.

3. Portability: the demand on the compiler is modest. No sophisticated analysis is required and performance tuning can be done in a straightforward manner using a

cost model for the relevant parameters. This allows any compiler to generate a well behaved and predictable implementation so that a program using the construct will behave consistently across platforms.

Although an implementation is not yet available in ZPL, these qualities leave little doubt that Mighty Scan will be a very useful language feature.

Chapter 7

Conclusions

> "**Mental Models** ... For computers and brains to be aware of something they must have an internal model of it – a representation, either digital or neurological. In the recent match, Mr. Kasparov kept honing his mental model of Deep Blue, developing a theory of how the machine worked...." [1]
>
> "It's very difficult to analyze the results of the match," Kasparov said. "I know what I did wrong. But I don't know what the computer did wrong or right. It's a mystery." [2]
> (World chess champion Gary Kasparov lost to IBM's Deep Blue machine in a six-game chess match, May 3-11, 1997.)

7.1 Contributions

In this thesis, my interest is in finding a solution to the problem of developing efficient parallel programs for data parallel applications. The solution must meet three requirements: scalability, portability and ease of use. I show that an appropriate *performance model* is the key component of a language that will precipitate these three qualities.

[1] In *Machine vs. Machine: Deep, Deeper, Deepest Blue* by George Johnson, New York Times, May 18, 1997.

[2] In *What Deep Blue Learned From Grandmasters* by Bruce Weber, New York Times, May 18, 1997.

The thesis makes the following contributions:

1. An experimental comparison and analysis of two general programming models.
2. An experimental comparison and analysis of two data parallel languages, HPF and ZPL, based on:
   - The performance model
   - A subset of the NAS benchmarks
3. A new high level data parallel abstraction that promotes scalability, portability, and ease of use: Mighty Scan.

7.2 Summary

We began in Chapter 1 with a discussion which introduces the concept of modeling in programming languages and outlines the three criteria for an effective parallel language. Chapter 2 presents experimental evidence to support the choice of a nonshared memory programming model as the base for a parallel language. Shared memory and nonshared memory versions of LU and WATER are compared through an analytical model and actual performance on 5 shared memory machines. In Chapter 3, we studied two data parallel languages that are based on a nonshared memory model, contrasting the language features and their implications. The *performance model*, the foundation of ZPL's design, emerges as the major difference between HPF and ZPL; therefore in Chapter 4 we formulate a methodology to quantify the benefit of the performance model. Two case studies using the array assignment and matrix multiplication clearly show that without a concise performance model, HPF cannot guarantee the users consistent and portable performance. In Chapter 5, we studied the performance of HPF and ZPL in three NAS benchmarks: EP, FT and MG. In addition to confirming the critical need for a performance model, the results indicate that converting legacy Fortran 77 programs to HPF will be very difficult. On the other hand, ZPL in each case studies shows consistent and

predictable performance. Finally in Chapter 6, we studied a number of parallel solutions to a common pattern of sequential computation. The analysis leads to a proposal for Mighty Scan, a new language abstraction that promises to be scalable, portable and easy to use.

Many factors influence the success of a parallel language, some of which may be unrelated to the actual merit of the language. In this thesis, the analysis has lead us to the performance model, but clearly there are other pragmatic issues that are no less important. While the results show serious weakness in HPF, it is important that we maintain the larger perspective and recognize the independent contributions of HPF and ZPL.

From a situation of incompatible parallel platforms and nonportable programs five years ago, HPF was able to gain the attention of major software vendors and be accepted as a standard. This is a difficult feat since the software industry is only willing to invest in conservative approaches and is not likely to consider any new unproven language. HPF's conservative approach includes preserving the original Fortran sequential programming model in the parallel environment. The resulting programming model, as we see, is muddled and does not preserve the sequential model nor capture enough information for the parallel model.

It is unfortunate that the first standard for data parallel language is handicapped by serious limitations, yet these compromises may be the necessary sacrifice to gain wide acceptance in the industry and support from the users.

In contrast, ZPL is insulated from legacy and other requirements unrelated to parallelism; this has enabled ZPL researchers to gain valuable insights into abstract concepts that have a major impact on the effectiveness of the language, among which is the performance model.

The current situation in parallel systems finds many parties with a high stake in HPF. Compiler vendors have staked their future in HPF, while some national labs and supercomputing centers have actively promoted the use of HPF. This thesis has been

critical of HPF; therefore it is quite likely to evoke defenses for HPF. A frequent argument is, "The current compilers are immature, new compilers with new optimizations will give better performance". In the ideal case, given infinite resources and infinite time, perhaps a compiler can be developed that requires no programming effort and provides optimal performance. In the present case however, the data shows that other factors are involved beside the performance. In fact, if the performance is the only goal, each HPF compiler we considered has shown to be able to achieve good performance in specific instances.

The implementations of HPF have allowed us to learn many lessons, and these are probably the most valuable contributions from HPF. It is therefore imperative that we recognize and understand the lessons so that we can build from the current state of the art. In this respect, it would be counter-productive to insist on the conformance to a standard that has serious limitations, but it would be equally grievous to hold HPF as the exemplary failure of parallel programming in general.

This thesis has identified the performance model as one important lesson, but other lessons should also be recognized.

The ready acceptance of HPF despite its limitations underscores the endurance of Fortran as a programming language. Computer scientists tend to deplore Fortran as obsolete in light of new programming concepts, models, compiler optimizations, etc. However, to a user in the scientific community, the computer and the language are no more than useful tools. The user will invest no more effort than necessary to obtain a satisfactory result. If Fortran has become a familiar fixture, perhaps retaining at least some of the syntax and semantics of Fortran in a new parallel language is beneficial. This is especially complementary considering that a new parallel language will likely devote a large part of the syntax and semantics to the conventional constructs such as assignment, if, sequential loops. A familiar sight as the first impression of a new language will contribute significantly toward gaining user acceptance.

As a case in point, consider Java: although it is a new language, it has some of the look and feel of C. This enables a new user to command a large part of the language

syntax immediately, leaving only the new constructs to be learned.

In the final analysis, perhaps the naming scheme is the most pragmatic factor. The ideal parallel language of the future may have "*Fortran*" as a part of its name - much to the dismay of computer scientists, yet it may bear little resemblance to the original Fortran language. Several lessons may have to be learned before this goal is reached, but this is the normal progress of technology. It is my hope that the work in this thesis will contribute to this progress.

7.3 Future works

The syntax and semantics for Mighty Scan have been proposed. An implementation in ZPL remains to be completed. The NAS benchmarks SP, BT, LU and FT are likely to benefit significantly from this new construct since each contains the type of sequential computation that is awkward to parallelize. Therefore, they are good candidates for testing and tuning the Mighty Scan construct.

The arrival of Java also introduces new, interesting opportunities for parallel programming. Despite the exaggerated level of publicity surrounding Java, there are some advantages that are worth considering. The object-oriented model can help to encapsulate some of the high level abstractions for a nonshared memory machine. The support for threads in the language allows parallel programming at an intrinsic level. The secure nature of Java proves to be attractive for financial applications, which incidentally are similar to scientific applications and can benefit from parallelism (e.g., PDE). More pragmatically, the momentum that Java is generating promises good performance and widespread availability in the future. In this case, Java may serve as a modern replacement for Fortran. One challenge for using Java is the shared memory model that the language adopts: we must find a way to incorporate a performance model if a parallel Java program is to run on a nonshared memory machine.

Bibliography

[Adams et al. 92] J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener. *Fortran 90 Handbook.* McGraw Hill, New York, NY, 1992.

[Agarwal et al. 94a] R. Agarwal, F. Gustavson, and M. Zubair. An efficient parallel algorithm for the 3-D FFT NAS parallel benchmark. In *Proceedings of SHPCC 1994*, pages 129–133. IBM Thomas J. Watson Research Center, 1994.

[Agarwal et al. 94b] R. Agarwal, F. Gustavson, and M. Zubair. A very high performance algorithm for NAS EP benchmark. In *High Performance Computing and Networking*, pages 164–169. IBM Thomas J. Watson Research Center, 1994.

[Agarwal et al. 95] R. Agarwal, B. Alpern, L. Carter, F. Gustavson, D. K. R. Lawrence, and M. Zubair. High performance parallel implementation of the NAS kernel benchmarks on the IBM SP2. *IBM Systems Journal*, 34(2):263–272, 1995.

[Alverson et al. 93] G. Alverson, W. Griswold, C. Lin, D. Notkin, , and L. Snyder. Abstractions for portable, scalable parallel programming computing. Technical Report UW-CSE-TR 93-12-09, University of Washington, Seattle, Wa 98195, December 1993.

[Amarasinghe & Lam 93] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 126–38. Computer System Laboratory, Stanford University, CA, USA, June 1993.

[Anderson & Snyder 91] R. Anderson and L. Snyder. A comparison of shared and non-shared memory models of parallel computation. In *Proceedings of IEEE*, pages 480–487. Dept of Computer Science and Engineering, University of Washington, Seattle, Wa, April 1991.

[Andre & Priol 92] F. Andre and T. Priol. Programming distributed memory parallel computers without explicit message passing. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, pages 90–97, May 1992.

[Annaratone & Ruhl 89] M. Annaratone and R. Ruhl. Performance measurements on a commercial multiprocessor running parallel code. In *Proceedings of 16th Annual International Symposium on Computer Architecture*, pages 307–314, Los Amitos, CA, May 1989. Swiss Federal Institute of Technology, Zurich, Switzerland, IEEE Computer Society Press.

[App 95] Applied Parallel Research. *XHPF User's Guide*, version 2.0 edition, January 1995.

[Ashcraft 91] C. Ashcraft. A taxonomy of distributed dense LU factorization methods. Technical Report ECA-TR-161, Engineering Computing and Analysis Technical Report, March 1991.

[Bagheri et al. 94] B. Bagheri, A. Ilin, and L. R. Scott. A comparison of distributed and shared memory scalable architectures. 1. KSR shared memory. In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 9–16, Los Alamitos, CA, May 1994. TCAMC, Houston Univ., TX, IEEE Computer Society Press.

[Bailey et al. 91] D. Bailey, E. Barszcz, J. barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Finebertg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA, March 1991.

[Bailey et al. 95] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmark 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[Bar-Noy & Kipnis 92] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proceedings of the 1992 ACM Symposium on Parallel Algorithms and Architectures*, pages 13–22, Yorktown Heights, NY, June 1992. IBM T. J. Watson Research Center.

[Baylor & Rathi 89] S. J. Baylor and B. D. Rathi. A study of the memory reference behavior of engineering/scientific applications in parallel processors. In *1989 International Conference on Parallel Processing*, pages I78–I82, Yorktown Heights, NY, 1989. IBM T. J. Watson Research Center.

[Benkner et al. 92] S. Benkner, B. Chapman, and H. Zima. Vienna Fortran 90. In *Proceedings of Scalable High Performance Computing Conference 1992*, pages 51–59. Department for Statistics and Computer Science, Vienna University, Austria, 1992.

[Bozkus et al. 94] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21:15–26, 1994.

[Bozkus et al. 95] Z. Bozkus, L. Meadows, S. Nakamoto, V. Schuster, and M. Young. Compiling High Performance Fortran. In *Proceedings ofthe Seventh SIAM Conference on Parallel Processing for Scientific Computing*. The Portland Group, Inc., 1995.

[Breit et al. 93] S. Breit, C. Pangali, and D. Zirl. Technical applications on the KSR-1: high performance and ease of use. In *COMPCON Spring 1993*, pages P303–310, Los Alamitos, CA, February 1993. Kendall Square Research Corp, Waltham, MA, IEEE Computer Society Press.

[Brooks et al. 91] E. Brooks, B. Gorda, K. Warren, and T. Welcome. Split-Join and message passing programming models on the BBN TC2000. In *International Conference on Parallel Processing*, pages II54–59. Lawrence Livermore National Laboratory, Livermore, CA, 1991.

[Brorsson 91] M. Brorsson. Local vs. global memory in the IBM RP3: Experiments and performance modeling. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 496–503. Department of Computer Engineering, Lund University, Sweden, December 1991.

[Burke 93] E. Burke. An overview of system software for the KSR-1. In *COMPCON Spring 1993*, pages P295–299, Los Alamitos, CA, February 1993. Kendall Square Research Corp, Waltham, MA, IEEE Computer Society Press.

[Byrd & Delagi 88] G. Byrd and B. Delagi. A performance comparison of shared variables versus message passing. In *The Third International Conference on Supercomputing*, volume 1, pages 1–7, May 1988.

[Chamberlain et al. 95] B. Chamberlain, S.-E. Choi, E. Lewis, C. Lin, L. Snyder, and D. Weathersby. The implementation of a machine-independent array language. University of Washington, 1995.

[Choi & Snyder 97] S.-E. Choi and L. Snyder. Quantifying the effects of communication optimizations. Technical Report UW-CSE-97-04-05, University of Washington, April 1997.

[Cox & Fowler 93] A. Cox and R. Fowler. Adaptive cache coherency for detecting migratory shared data. *Computer Architecture News*, 21(2):98–108, May 1993.

[Crowther et al. 85] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance measurements on a 128-node Butterfly parallel processor. In *International Conference on Parallel Processing*, pages 531–540, 1985.

[Cytron 86] R. Cytron. Doacross: Beyond Vectorization for Multiprocessors. In *International Conference on Parallel Processing*, pages 836–844. IBM T. J. Watson Research Center, Yorktown Heights, NY, 1986.

[Darema-Rogers et al. 87] F. Darema-Rogers, G. Pfister, and K. So. Memory access patterns of parallel scientific programs. In *Proceedings of SIGMETRICS*, pages 46–58, 1987.

[Dig 95] Digital Equipment Corporation. *HPF tutorial*, version 1.1 edition, July 1995.

[Eggers & Katz 88] S. Eggers and R. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of 15th Annual International Symposium on Computer Architecture*, pages 373–382, Los Alamitos, CA, June 1988. Dept of Electrical Engineering & Computer Science, University of California, Berkeley, CA, IEEE Computer Society Press.

[Eigenmann et al. 91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect benchmark programs. In *The Fourth Workshop on Languages and Compilers for Parallel Computing*, pages g1–g19, August 1991.

[Emrath et al. 89] P. Emrath, D. Padua, and P. Yew. Cedar architecture and its software. In *Proceedings of the Twenty Second Annual Hawaii International Conference on System Sciences. Vol.I: Architecture Track*, pages 306–315. Center for Supercomputing Research & Development, Illinois Univisity, Urbana, IL, January 1989.

[Felten 93] E. Felten. *Protocol Compilation: High Performance Communication for Parallel Program*. PhD dissertation, University of Washington, Seattle, Wa, December 1993.

[Forum 93] H. P. F. Forum. HPF language specification version 1.0. Technical Report CRPC-TR92225, Rice University, May 1993.

[Forum 96] H. P. F. Forum. HPF language specification version 2.0. Technical report, Rice University, October 1996.

[Frank et al. 93] S. Frank, H. Burkhardt, and J. Rothnie. The KSR-1: bridging the gap between shared memory and MPPs. In *COMPCON Spring 1993*, pages

285–294, Los Alamitos, CA, February 1993. Kendall Square Research Corp, Waltham, MA, IEEE Computer Society Press.

[Franke et al. 94] H. Franke, P. Hochschild, P. Pattnaik, and M. Snir. An efficient implementation of MPI. Technical Report RC 19493, IBM T. J. Watson Research Center, Yorktown Heights, NY, March 1994.

[Friedman et al. 95] R. Friedman, J. Levesque, and G. Wagenbreth. Fortran parallelization handbook. Technical report, Applied Parallel Research, Sacramento, CA, April 1995.

[Garber 93] M. Garber. The TC2000 system - a large scale shared memory multiprocessor. *International Journal of High Speed Computing*, 5(3):475–490, 1993.

[Gharachorloo et al. 92] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding memory latency using dynamic scheduling in shared memory multiprocessors. *Computer Architecture News*, 20(2):22–33, May 1992.

[Gifford 87] P. Gifford. Symmetry: a bus-based multiprocessor with copy-back caches. In *Proceedings of the 1987 IEEE International Conference on Computer Design*, page 62, Los Amitos, CA, October 1987. Sequent Computer System, Beaverton, OR, IEEE Computer Society Press.

[Gropp 93a] W. Gropp. Early experiences with the IBM SP-1. Technical Report ANL/MCS-TM-177, Argonne National Laboratory, Argonne, IL, 1993.

[Gropp 93b] W. Gropp. Early experiences with the IBM SP1 and the high-performance switch. Technical Report ANL-93/41, Argonne National Laboratory, Argonne, IL, November 1993.

[Gupta et al. 91] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W. Weber. Comparative evaluation of latency reducing and tolerating techniques. *Computer Architecture News*, 19(3):254–263, May 1991.

[Gupta et al. 94] M. Gupta, E. Schonberg, and H. Srinivasan. A unified data-flow framework for optimizing communication. In *7th International Workshop Proceedings on Languages and Compilers for Parallel Computing.*, pages 266–282. IBM Thomas J. Watson Research Center, August 1994.

[Gupta et al. 95] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching, and T. Ngo. An HPF compiler for the IBM SP2. In *Supercomputing 1995*, San Diego, December 1995. IBM T. J. Watson Research Center, IEEE.

[Hariri et al. 93] S. Hariri, J. Park, F.-K. Yu, M. Parashar, and G. C. Fox. A Message Passing Interface for parallel and distributed computing. In *Proceedings of*

the 2nd International Symposium on High Performance Distributed Computing, pages 84–91, Los Alamitos, CA, July 1993. Northeast Parallel Architectures Center, IEEE Computer Society Press.

[Harris et al. 95] J. Harris, J. Bircsak, R. Bolduc, J. Diewald, I. Gale, N. Johnson, S. Lee, A. Nelson, and C. Offner. Compiling High Performance Fortran for distributed-memory systems. *Digital Technical Journal*, 7(3):5–38, 1995.

[Heywood & Ranka 92] T. Heywood and S. Ranka. A practical hierarchical model of parallel computation. *Journal of Parallel and Distributed Computing*, 16:212–232, 1992.

[Hiranandani et al. 94] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluating compiler optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21:27–45, 1994.

[Hochschild 93] P. Hochschild. *EUIH: An Experimental EUI Implementation.* IBM T. J. Watson Research Center, Yorktown Heights, NY, version 1.06.3 edition, September 1993.

[Hockney & Carmona 92] R. Hockney and E. Carmona. Comparison of communications on the Intel iPSC/860 and Touchstone Delta. *Parallel Computing*, 18:1067–1072, 1992.

[Hockney 94] R. W. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, March 1994.

[IBM 95] IBM Thomas J. Watson Research Center. *UTE User's Guide for IBM SP System*, version 0.92 edition, September 1995.

[Int 91a] Intel Supercomputer System Division, Beaverton, Or. *A Touchstone Delta System Description*, February 1991.

[Int 91b] Intel Supercomputer System Division, Beaverton, Or. *Touchstone Delta System User's Guide*, 312125-001 edition, October 1991.

[Int 93] Intel Supercomputer System Division, Beaverton, Or. *Paragon User's Guide*, 312489-002 edition, 1993.

[Jeremiassen & Eggers 95] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–88. University of Washington, Seattle, WA, 1995.

[Joe & Hennessy 94] T. Joe and J. Hennessy. Evaluating the memory overhead required for COMA architectures. In *Proceedings the 21st Annual International Symposium on Computer Architecture*, pages 82–93, Los Alamitos, CA, April 1994. Computer System Lab, Stanford University, CA, IEEE Computer Society Press.

[Joi 95] Joint Institute For Computational Science. *A Beginner's Guide to the Maspar MP-2*, February 1995.

[Karp 87] A. Karp. Programming for parallelism. *Computer*, pages 43–56, May 1987.

[Kendall Square Research 92] Kendall Square Research. KSR technical summary. Technical report, 1992.

[Klaiber & Levy 94] A. Klaiber and H. Levy. A comparison of message passing and shared memory architectures for data parallel programs. In *Proceedings of 21st Annual International Symposium on Computer Architecture*, pages 94–105, Los Alamitos, CA, April 1994. Department of Computer Science and Engineering, University of Washington, Seattle, WA, IEEE Computer Society Press.

[Konicek et al. 91] J. Konicek, T. Tilton, A. Veidenbaum, C. Zhu, E. Davidson, R. Downing, M. Haney, M. Sharma, P. Yew, P. Farmwald, D. Kuck, D. Lavery, R. Lindsey, D. Pointer, J. Andrews, T. Beck, T. Murphy, S. Turner, and N. Warter. The organization of the Cedar system. In *International Conference on Parallel Processing*, pages I49–56. Center for Supercomputing Research and Development, University of Illinois, IL, 1991.

[Kuck et al. 93] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C. Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, J. Andrews, and S. Turner. The Cedar system and an initial performance study. In *20th Annual International Symposium on Computer Architecture*, pages 213–223. Center for Supercomputing Research and Development, Illinois University, Urbana, IL, May 1993.

[LeBlanc 86] T. LeBlanc. Shared-memory versus message-passing in a tightly-coupled multiprocessor: A case study. In *International Conference on Parallel Processing*, pages 463–466, 1986.

[Leiserson 85] C. Leiserson. Fat-Trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, October 1985.

[Lenoski et al. 92a] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, March 1992.

[Lenoski et al. 92b] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: implementation and performance. *Computer Architecture News*, 20(2):92–105, May 1992.

[Lenoski et al. 93] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.

[Leslie 90] V. Leslie. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[Lewis et al. 94] J. Lewis, D. Payne, and R. van de Geijn. Matrix-vector multiplication and conjugate gradient algorithms on distributed memory computers. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 542–550. University of Texas at Austin, 1994.

[Li & Hudak 89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):463–466, November 1989.

[Lim & Yew 91] H. Lim and P. Yew. Parallel program behavioural study on a shared memory multiprocessor. In *1991 International Conference on Supercomputing*, pages 386–395, New York, NY, June 1991. Center for Supercomputing Research & Development, Illinois University, Urbana Champaign, IL, ACM.

[Lin & Snyder 90] C. Lin and L. Snyder. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, volume II, pages 163–170, 1990.

[Lin & Snyder 93] C. Lin and L. Snyder. ZPL: An array sublanguage. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 96–114. 1993.

[Lin 91] C. Lin. Portable parallel programming: Cross machine comparisons for Simple. In *Proceedings of the 5th SIAM Conference on Parallel Processing*, 1991.

[Lin 92] C. Lin. *The Portability of Parallel Programs Across MIMD Computers.* PhD dissertation, University of Washington, Seattle, Wa, December 1992.

[Lin 94] C. Lin. ZPL language reference manual. Technical Report UW-CSE-TR 94-10-06, University of Washington, October 1994.

[Lin et al. 93] M. Lin, R. Tsang, and D. Du. Performance characteristics of the Connection Machine hypertree network. *Journal of Parallel and Distributed Computing*, 19(3):245–254, November 1993.

[Lin et al. 95] C. Lin, L. Snyder, R. Anderson, B. Chamberlain, S.-E. Choi, G. Forman, E. Lewis, and D. Weathersby. ZPL vs. HPF: A comparison of performance and programming style. Technical Report UW-CSE-95-11-05, University of Washington, Seattle, Wa 98195, November 1995.

[Lovett & Thakkar 88] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. Technical report, Sequent Computer Systems, 15450 SW Koll Parkway, Beaverton, Or, 1988.

[Naik 94] V. Naik. Performance of NAS parallel application-benchmarks on IBM SP1. In *Proceedings of the Scalable High Performance Computing Conference*, pages 121–128. IBM Thomas J. Watson Research Center, 1994.

[Naik 95a] V. Naik. Performance of NAS parallel benchmark LU on IBM SP system. Technical Report RC20046, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, March 1995.

[Naik 95b] V. Naik. A scalable implementation of the NAS parallel benchmark BT on distributed memory systems. *IBM Systems Journal*, 34(2):273–291, 1995.

[Naik et al. 93] N. Naik, V. Naik, and M. Nicoules. Parallelization of a class of implicit finite difference schemes in computational fluid dynamics. *International Journal of High Speed Computing*, 5(1):1–50, 1993.

[Ngo & Snyder 92] T. Ngo and L. Snyder. On the influence of programming models on shared memory computer performance. In *Proceedings of the 1992 Scalable High Performance Computing Conference*, pages 284–291. Dept of Computer Science and Engineering, University of Washington, Seattle, Wa, May 1992.

[Papadopoulos & Culler 90] G. Papadopoulos and D. Culler. Monsoon: an Explicit Token-Store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–91, Los Alamitos, CA, May 1990. Laboratory for Computer Science, Masachusetts Institute of Technology, IEEE Computer Society Press.

[Picano et al. 92] S. Picano, E. Brooks, and J. E. Hoag. Quantifying programming costs vs. scalable cache coherency on a large scale shared memory multiprocessor. Technical report, Lawrence Livermore National Laboratory, Livermore, CA 94550, August 1992.

[Pinkston & Baylor 91] T. M. Pinkston and S. J. Baylor. Parallel processor memory reference analysis: examining locality and clustering potential. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 513–518, Philadelphia, PA, March 1991. IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, SIAM.

[Ponnusamy et al. 92] R. Ponnusamy, A. Choudhary, and G. Fox. Communication overhead on the CM-5: an experimental performance evaluation. In *The Fourth Symposium on the Frontiers of Massively Parallel Computation: Frontiers 1992*, pages 108–115, Los Alamitos, CA, October 1992. Syracuse Univiversity, NY, IEEE Computer Society Press.

[Ponnusamy et al. 93] R. Ponnusamy, R. Thakur, A. Choudhary, K. Velamakanni, Z. Bozkus, and G. Fox. Experimental performance evaluation of the CM-5. *Journal of Parallel and Distributed Computing*, 19(3):192–202, November 1993.

[Presberg 96] D. Presberg. Comparison of 3 hpf compilers for the ibm sp. In *NHSE Review, http://www.crpc.rice.edu/NHSEreview/HPF*. Cornell Theory Center, 1996.

[Qin & Baer 97] X. Qin and J.-L. Baer. On the Use and Performance of Explicit Communication Primitives in Cache-coherent Multiprocessor Systems. In *High Performance Computer Architecture*, pages 182–193. University of Washington, 1997.

[Robert 90] Y. Robert. *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*. Halsted Press, 1990.

[Saini & Bailey 95] S. Saini and D. Bailey. NAS parallel benchmark result 12/95. Technical Report NAS-95-021, NASA Ames Research Center, Moffett Field, CA, December 1995.

[Saini 95] S. Saini. NAS experiences of porting CM Fortran codes to HPF on IBM SP2 and SGI Power Challenge. Technical Report NAS-95-010, NASA Ames Research Center, Moffett Field, CA, April 1995.

[Saphir et al. 95] W. Saphir, A. Woo, and M. Yarrow. NAS parallel benchmark 2.1 results: 8/96. Technical Report NAS-96-010, NASA Ames Research Center, Moffett Field, CA, December 1995.

[Singh & Hennessy 91] J. Singh and J. Hennessy. Data locality and memory system performance in the parallel simulation of ocean eddy currents. In *Proceedings of the Second Symposium on High Performance Computing*, pages 43–57, Amsterdam, Netherlands, October 1991. Computer System Laboratory, Stanford University, CA, North Holland.

[Singh et al. 92] J. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[Singh et al. 93] J. Singh, J. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: methodology and examples. *Computer*, 26(7):42–50, July 1993.

[Singh et al. 94] J. P. Singh, E. Rothberg, and A. Gupta. Modeling communication in parallel algorithms: A fruitful interaction between theory and systems? Technical report, Stanford University, 1994.

[Snyder 86] L. Snyder. Type architecture, shared memory and the corollary of modest potential. *Annual Review of Computer Science*, 1:289–318, 1986.

[Snyder 94] L. Snyder. A ZPL programming guide. Technical Report UW-CSE-TR 94-12-02, University of Washington, December 1994.

[Snyder 95] L. Snyder. Experimental validation of models of parallel computation. In A. Hofmann and J. van Leeuwen, editors, *Lecture Notes in Computer Science*, volume Special Volume 1000, pages 78–100. Springer-Verlag, 1995.

[Stark & Beris 91] S. Stark and A. Beris. LU decomposition optimized for a parallel computer with a hierarchical distributed memory. In *The 1991 MPCI Yearly Report: The Attack of the Killer Micros*, pages 127–132, March 1991.

[Stunkel et al. 94a] C. Stunkel, M. Denneau, B. Nathanson, D. Shea, P. Hochschild, M. Tsao, B. Abali, D. Joseph, and P. Varker. Architecture and implementation of Vulcan. In *Proceedings of Eighth International Parallel Processing Symposium*, pages 268–274. IBM T. J. Watson Research Center, Yorktown Heights, NY, April 1994.

[Stunkel et al. 94b] C. Stunkel, D. Shea, D. Grice, P. Hochschild, and M. Tsao. The SP1 high-performance switch. In *Proceedings of the 1994 Scalable High Performance Computing Conference*, pages 150–157. IBM T. J. Watson Research Center, Yorktown Heights, NY, May 1994.

[Tadjbakhsh 93] S. Tadjbakhsh. *IBM AIX Parallel Environment Parallel Programming Reference*. IBM Information Development, Kingston, NY, sh26-7228-00 edition, August 1993.

[Thakkar 87] S. Thakkar. A performance analysis of a shared memory multiprocessor. In *Proceedings of the 1987 IEEE International Conference on Computer Design*, Los Amitos, CA, October 1987. Sequent Computer System, Beaverton, OR, IEEE Computer Society Press.

[Thakkar et al. 88] S. Thakkar, P. Gifford, and G. Fielland. The Balance multiprocessor system. *IEEE Micro*, pages 57–69, February 1988.

[Thekkath & Eggers 94] R. Thekkath and S. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Proceedings of 21st Annual International Symposium on Computer Architecture*, pages 176–186, Los Alamitos,

CA, April 1994. Department of Computer Science and Engineering, University of Washington, Seattle, WA, IEEE Computer Society Press.

[Thi 93a] Thinking Machine Corporation, Cambridge, MA. *CMMD Reference Manual*, version 3.0 edition, May 1993.

[Thi 93b] Thinking Machine Corporation, Cambridge, MA. *CMMD User's Guide*, version 3.0 edition, May 1993.

[Thi 93c] Thinking Machine Corporation, Cambridge, MA. *Connection Machine CM-5 Technical Summary*, November 1993.

[Thi 94] Thinking Machines Corporation. *CM Fortran Programming Guide*, version 2.2 edition, October 1994.

[van de Geijn & Watts 95] R. van de Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. Technical Report TR-95-13, University of Texas, Austin, Texas, April 1995.

[Windheiser et al. 93] D. Windheiser, E. Boyd, E. Hao, S. Abraham, and E. Davidson. KSR-1 multiprocessor: Analysis of latency hiding techniques in a sparse solver. In *Proceedings of Seventh International Parallel Processing Symposium*, pages 454–461, Los Alamitos, CA, April 1993. Dept of Electrical Engineering & Computer Science, Michigan University, Ann Arbor, MI, IEEE Computer Society Press.

[Wu & Benveniste 94] E. Wu and C. Benveniste. *A Unified Trace Environment for SPx Systems*. IBM T. J. Watson Research Center, Yorktown Heights, NY, version 0.4 edition, March 1994.

[Zhang 91] X. Zhang. System effects of interprocessor communication latency in multicomputers. *IEEE Micro*, 11(2):12–15, 52–55, April 1991.

[Zhang et al. 94a] X. Zhang, Y. Yan, and K. He. Evaluation and measurement of multiprocessor latency patterns. In *Proceedings of Eighth International Parallel Processing Symposium*, pages 845–852, Los Alamitos, CA, April 1994. High Performance Computing & Software Lab, Texas Univ., San Antonio, TX, IEEE Computer Society Press.

[Zhang et al. 94b] X. Zhang, Y. Yan, and K. He. Latency metric: An experimental method for measuring and evaluating parallel program and architecture scalability. *Journal of Parallel and Distributed Computing*, 1994.

Appendix A

Performance data

A.1 LU on shared-memory machines

The following tables show the execution times in seconds for LU on 5 shared-memory machines. This is the data for chapter 2.

Table A.1: Performance (seconds) of LU Decomposition on 5 shared memory machines.

LU on Sequent (seconds)						
	200×200		300×300		512×512	
processor	sm	nsm	sm	nsm	sm	nsm
1	45.370148	45.619545	151.671639	152.84721	749.710614	757.425652
2	22.764425	23.165471	76.047993	77.736939	374.954634	379.982583
4	11.463419	12.061735	38.075155	39.535023	187.550758	192.953820
6	7.812354	8.319552		26.950458		133.463661
8	5.931265	6.458318	19.382059	20.741765	94.532781	98.694475
10	4.880708	5.255464		16.906972		79.909908
12	4.171789	4.628903		14.475015		67.285550
14	3.718850	4.111252		12.757039		58.894898
16	3.386981	3.789275	10.242270	11.301436	48.279209	51.695656

LU on KSR (seconds)						
	200×200		300×300		512×512	
processor	sm	nsm	sm	nsm	sm	nsm
1	4.963139	4.128038	16.674863	13.771776	81.422266	67.440069
2	2.917240	2.098688	9.495537	6.945457	45.288623	33.975297
4	1.531988	1.113727	4.875464	3.540075	23.306514	17.281725
8	0.890201	0.668275	2.666797	1.980642	12.259547	8.845870
16	0.643920	0.562256	1.690843	1.288997	6.928109	4.907069
24	0.659858	0.579727	1.471892	1.292831	5.335590	4.044066
32	0.895496	0.598683	1.786207		5.170310	3.863252

LU on Cedar (seconds)						
	200×200		300×300		512×512	
processor	sm	nsm	sm	nsm	sm	nsm
1	64.64052	44.43229	216.85811	150.38560	1074.41066	757.65471
2	32.44478	23.20331	108.53429	76.26080	539.33373	378.21246
4	16.29816	12.05089	55.92126	39.14370	269.30365	192.06901
8	8.53105	6.85880	27.66585	21.52057	137.76543	100.01388
12		5.16418		15.46952	92.03920	68.81427
16	4.93295	4.44266	14.83814	12.22920	72.05867	54.51414
20		4.28967		10.93311		49.77601
24	4.4094	4.26784	11.24298	10.21729	51.33575	40.55990
28		4.25487		10.06788		36.90922
32	4.6288	4.37602	10.68505	10.32745	40.15609	35.13364

LU on Butterfly T2000 (seconds)						
	200×200		300×300		512×512	
processor	sm	nsm	sm	nsm	sm	nsm
1	21.456812		71.462568			
2	10.892948	4.053568	35.883065	13.540942	182.903653	67.573137
4	5.647862	2.165374	18.496054	6.936486	94.217712	34.356636
8	3.498626	1.251953	11.105882	3.883010	55.118757	17.783904
16	3.317192	.881832	9.992702	2.307044	45.819111	9.684133
24	3.598184	.937469	10.151217	2.122591	45.111404	7.777608
32	3.948025	.994012	10.779631	2.240521	45.755199	6.689305
48	4.436488	1.101099	11.699838	6.487641	50.215238	7.035376

LU on DASH (seconds)						
	200×200		300×300		512×512	
processor	sm	nsm	sm	nsm	sm	nsm
1	3.209	3.179	10.919	10.727	67.014	54.832
2	1.635	1.674	5.786	5.418	32.372	27.428
4	.857	.873	2.877	2.768	15.863	13.813
8	.559	.499	1.695	1.555	8.088	7.150
16	.369	.350	1.083	.973	4.729	4.015
32	.418	.390	.942	.953	3.709	2.914

A.2 WATER on shared-memory machines

The following tables show the execution times in seconds for WATER on 5 shared-memory machines. This is the data for chapter 2.

Table A.2: Performance (seconds) of WATER on 5 shared memory machines.

WATER on Sequent (seconds)						
	96 mols		288 mols		512 mols	
processor	sm	nsm	sm	nsm	sm	nsm
1	56.230536	56.302039	455.615188	456.720134	1405.776632	1410.271537
2	28.796820	29.019462	230.486377	231.114972	715.609000	717.309184
4	14.695737	14.982332	117.139749	117.536687	367.452312	368.393325
6	9.879172	10.256695	77.890070	78.778194		
8	7.428865	7.829909	59.098132	59.773076	186.251235	205.417192
12	4.951295	5.277917	40.699293	41.231677		
16	3.744649	4.063647	31.140790	31.444728	96.469742	97.108076
18			28.161897	28.425073		

WATER on KSR (seconds)						
	96 mols		288 mols		512 mols	
processor	sm	nsm	sm	nsm	sm	nsm
1	6.28	6.12	48.68	47.16	148.50	143.20
2	3.36	3.22	25.14	24.36	74.80	72.46
4	1.78	1.66	13.26	12.40	39.64	36.58
8	.92	.90	6.90	6.44	20.36	18.62
16	.50	.50	3.68	3.46	10.84	9.98
24	.42	.42	2.62	2.46	8.78	6.54
32	.42	.36	2.08	1.92	5.70	5.32

WATER on CEDAR (seconds)						
	96 mols		288 mols		512 mols	
processor	sm	nsm	sm	nsm	sm	nsm
1	74.933640	76.402960	599.506110	575.788210		
2	38.552850	38.195740	304.091360	289.909760		
4	19.530560	19.900750	152.750320	148.831810		
8	9.870800	10.748890	77.355800	84.695670	238.718480	233.007470
12	6.689800	7.733460	52.907180	77.367830		
16	5.169280	6.128690	40.013000	42.731850	122.587380	126.089350
24	3.500040	4.561320	30.283210	29.494360	107.214710	83.444320
32	3.174200	4.653100	21.239410	24.572020	67.686400	68.995810

WATER on Butterfly T2000 (seconds)						
	96 mols		288 mols		512 mols	
processor	sm	nsm	sm	nsm	sm	nsm
1	24.378128	16.910681	191.955684	129.875652	1754.425691	1437.595238
2			98.024568	70.422117		
4	6.465225	4.749529	50.213944	34.703714	150.397842	102.121784
8	3.312763	2.647135	25.525649	18.376480	76.523083	52.861218
12			17.925968	13.200817		
16	1.749591	1.639289	13.759438	10.408390	40.780034	29.573802
24			9.936112	8.186353		
32	1.163515	1.350450	8.249899	7.406816	24.293119	20.896717
48	1.126044	1.301894	7.413884	6.868895		
64					20.218286	19.550439
72			7.508387	7.251650		
96	1.516591	2.306298	7.996893	8.414836	16.569527	19.502120

WATER on DASH (seconds)						
	96 mols		288 mols		512 mols	
processor	sm	nsm	sm	nsm	sm	nsm
1	3.597	3.606	28.977	29.310	90.809	90.780
2	1.860	1.902	14.604	14.750	46.014	45.630
4	.960	1.001	7.511	7.620	24.075	23.630
8	.531	.601	3.935	4.140	12.569	12.420
16	.270	.381	2.091	2.330	6.441	6.883
24	.220	.280	1.501	1.880	5.527	
32	.150	.260	1.140	1.550	3.482	5.243

A.3 HPF and ZPL programs on nonshared-memory machine

The following tables show statistics on the SP2 parallel platforms and the execution time in seconds for the HPF and ZPL programs. This data is used in Chapter 4 and 5.

Table A.3: Characteristics of the IBM SP2 used for the cross-compiler comparison

	SP2
Site	Cornell Theory Center
Operating System	AIX 4.1.4
IBM HPF compiler	Version 1, Release 1
APR HPF compiler	Version 2.0
PGI HPF compiler	Version 2.0.2
ZPL compiler	Version 1.0
Communication lib	MPI (IBM)
Nodes	512
FLOPS/node	266 MFLOPS
Memory/node	128-2048 MB
Topology	MIN
Bandwidth	48MB/sec
Latency	40 usec

Table A.4: Performance (seconds) of Matrix Multiplication on the IBM SP2: 2000×2000

processor	compiler	DO loop	HPF opt	Cannon	SUMMA
16	ZPL	n/a	n/a	138.06	35.16
	IBM	10.62	21.00	111.38	11.51
	APR	56.14	24.56	nomem	20.69
	PGI	timeout	72.11	109.99	25.55
64	ZPL	n/a	n/a	44.87	10.08
	IBM	3.09	14.38	37.39	4.23
	APR	59.95	18.30	326.12	7.96
	PGI	timeout	12.92	37.43	8.44

Table A.5: Performance (seconds) of NAS 2.1 benchmarks on the IBM SP2: EP, FT, MG.

Embarrassingly Parallel						
	compiler	p=1	4	8	16	32
class S	ZPL	n/a	1.06	0.52	0.32	n/a
	APR do		1.40	0.81	0.44	
	IBM do		4.94	4.92	4.95	
	PGI do		4.98	5.00	4.97	
class A	ZPL	n/a	n/a	136.49	68.48	35.26
	APR do			159.33	79.93	47.83
	IBM F90			417.12	208.86	105.39
	PGI F90			419.40	210.16	107.10

MultiGrid						
	compiler	p=1	4	8	16	32
class S	MPI	0.15	0.09	0.06	n/a	n/a
	ZPL	1.47	0.60	0.42		
	APR do	28.58	181.76	250.59		
	IBM do	4.10	17.79	23.39		
	PGI do	48.08	274.14	385.12		
	APR F90	42.07	46.70	53.67		
	IBM F90	1.30	0.76	0.76		
	PGI F90	1.82	1.07	1.89		
class A	MPI	n/a	n/a	7.40	4.10	2.30
	ZPL			40.50	22.82	11.25
	APR F90			timeout	timeout	timeout
	IBM F90			140.09	79.99	46.52
	PGI F90			43.19	32.70	23.17

Fourier Transform						
	compiler	p=1	4	8	16	32
class S	MPI	4.70	1.49	0.83	n/a	n/a
	ZPL	26.96	7.11	3.78		
	APR do	100.63	205.16	timeout		
	IBM do	32.31	84.41	101.47		
	PGI do	175.26	timeout	timeout		
	APR F90	6.42	3.80	5.22		
	PGI F90	11.78	4.97	2.91		
	IBM F90	12.66	17.23	18.37		
		13.21	10.43	10.18		
class A	MPI	n/a	n/a	26.00	13.50	8.57
	ZPL			segfault	129.38	47.32
	APR F90			1607.16	37.26	29.96
	IBM F90			timeout	353.31	timeout
	PGI F90			321.28	61.30	32.56

Vitae

Ton Anh Ngo

Dept. of Computer Science & Engineering	IBM Thomas J. Watson Research Center
Box 352350	H1-J20
University of Washington	P.O. Box 704
Seattle, WA 98195	Yorktown Heights, NY 10598
914-784-7935	914-784-7935
tango@cs.washington.edu	tango@watson.ibm.com

Summary

Current Status: Ph.D. Candidate, University of Washington
Engineer, IBM T. J. Watson Research Center

Research Interest: Parallel architecture, Parallel programming model,
Latency hiding techniques

Graduate: **Ph.D., Computer Science,** University of Washington, expected 1997
M.S., Computer Science, 1992, University of Washington
M.S., Electrical Engineering, 1986, Florida Institute of Technology

Undergraduate: **B.S., Electrical Engineering,** 1982, Georgia Institute of Technology,
Highest Honor

Work experiences

1987-current IBM T. J. Watson Research Center, Yorktown Heights, NY:

1996-current	Java multithread debugger, Java parallel programming on the SP2
1995-1996	Implementation of an HPF compiler for the SP2 parallel computer
1989-1990	Development of PV, parallel program visualization
1987-1989	Hardware designs for the RP3 parallel computer: cache, performance monitor and floating point interface

1982-1987 IBM System Product Division, Boca Raton, FL:
Simulation, validation of Series/1 16-bits microprocessor

Publications

1. Portable Performance of Data Parallel Languages, with Lawrence Snyder, Bradford Chamberlain (to appear in Supercomputing 1997).

2. SPMD Programming in Java, with Susan Flynn Hummel, Harini Srinivasan. *Concurrency Practice and Experience,* Vol. 9(6), pp 621-631, June 1997.

3. An HPF Compiler for the IBM SP2, with M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.Y. Wang, W. Ching. *Supercomputing 1995,* San Diego, December 1995.

4. Data Locality On Shared Memory Computers Under Two Programming Models, with Lawrence Snyder. UW TR number: 93-06-08, IBM Research Report: RC 19082, June 1993.

5. On the Influence of Programming Models on Shared Memory Computer Performance with Lawrence Snyder. *Scalable High Performance Computing Conference,* April 1992.

6. The RP3 Program Visualization Environment, with Douglas Kimelman. *IBM Journal of Research and Development,* Vol 35, No 5/6, pp 635-651, September - November 1991.

7. Initial Experience with RP3 Performance Monitoring, with W. Brantley, L Brochard, A. Bolmarcich, H. Chang, K. McAuliffe. *Journal of High Speed Computing,* Vol 1, No 4, 1989.

8. RP3 Performance Monitoring Hardware, with William Brantley, Kevin McAuliffe. *Instrumentation for Future Parallel Computing System,* Addison-Wesley, 1989.

Personal Biography

Birthdate:	March 14, 1960	Sex:	Male
Home address:	115 Mitchell Road	Marital status:	married
	Somers, NY 10589	Spouse:	Hue Nguyen Ngo, D.O.
	914-277-1082	Children:	2 (ages 12 and 5)
Foreign language:	Vietnamese, French		
Interests:	camping, skiing, family activities		

Honors:

1978, High School Valedictorian
1978-1990, Dean's list, Georgia Institute of Technology
1982, BS degree with highest honor
1982, Member of Eta Kappa Nu, Tau Beta Sigma
1990, IBM Resident Study program
1989, 1996, IBM Research Division Awards

References

Professor Lawrence Snyder
Department of Computer Science & Engineering
Box 352350
University of Washington
Seattle, WA 98195
snyder@cs.washington.edu
206-543-9265

Fran Allen, Ph.D.
IBM Fellow
H1-D14
PO Box 704
Yorktown Heights, NY 10598
allen@watson.ibm.com
914-784-7518