

University of Montana

## ScholarWorks at University of Montana

---

Graduate Student Theses, Dissertations, &  
Professional Papers

Graduate School

---

1999

### Partial parallelization of VMEC system

Mei Zhou

*The University of Montana*

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

**Let us know how access to this document benefits you.**

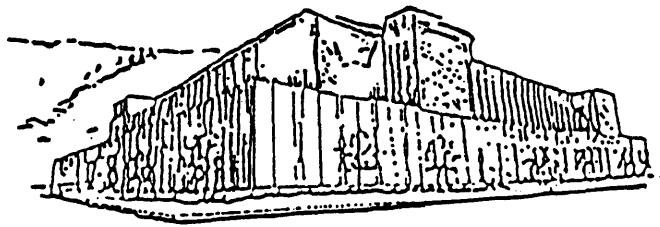
---

#### Recommended Citation

Zhou, Mei, "Partial parallelization of VMEC system" (1999). *Graduate Student Theses, Dissertations, & Professional Papers*. 5507.

<https://scholarworks.umt.edu/etd/5507>

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact [scholarworks@mso.umt.edu](mailto:scholarworks@mso.umt.edu).



Maureen and Mike  
**MANSFIELD LIBRARY**

The University of **MONTANA**

---

Permission is granted by the author to reproduce this material in its entirety,  
provided that this material is used for scholarly purposes and is properly cited in  
published works and reports.

*\*\* Please check "Yes" or "No" and provide signature \*\**

Yes, I grant permission

No, I do not grant permission

Author's Signature     *mi Zhou*    

Date     5/20/99    

Any copying for commercial purposes or financial gain may be undertaken only with  
the author's explicit consent.

**Partial Parallelization of VMEC System**

by

Mei Zhou

Presented in partial fulfillment of the requirements

for the degree of

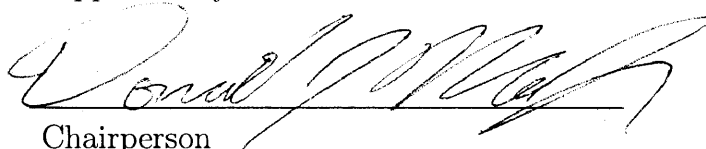
Master of Science

in Computer Science

The University of Montana-Missoula

May 1999

Approved by:



Chairperson



Dean, Graduate School

5-21-99

Date

UMI Number: EP40971

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

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



UMI EP40971

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 - 1346

Mei Zhou, M.S., May 1999

Computer Science

Partial Parallelization of VMEC System

Director: Donald J. Morton, Jr.



The VMEC (Variational Moments Equilibrium Code) is ported to a Cray T3E parallel computer system. Part of the code is parallelized using HPF ( High Performance Fortran). Parallel processing concepts and important HPF features are reviewed. The two steps in improving VMEC's performance are described. First, array operations in Fortran 90 are used to optimize the code. Then, data mapping and parallelism features of HPF are used to parallelize two subroutines of VMEC. Finally, testing results are presented and analyzed.

# TABLE OF CONTENTS

<b>ABSTRACT</b> .....	<b>ii</b>
<b>ACKNOWLEDGMENTS</b> .....	<b>vi</b>
<b>1 Introduction</b> .....	<b>1</b>
<b>2 Parallel Processing</b> .....	<b>3</b>
2.1 Parallel Computers .....	3
2.2 Parallel Computations .....	4
2.2.1 Data Parallelism .....	5
2.2.2 Shared Memory .....	5
2.2.3 Message Passing .....	6
2.3 Performance Issues .....	7
<b>3 High Performance Fortran</b> .....	<b>8</b>
3.1 Basics of High Performance Fortran .....	8
3.1.1 Fortran 90 .....	9
3.1.2 Compiler Directives .....	10
3.1.3 Parallelism Features .....	11
3.2 Data Mapping .....	12
3.2.1 DISTRIBUTE directive .....	13
3.2.2 ALIGN directive .....	13
3.2.3 TEMPLATE directive .....	14
3.2.4 PROCESSOR directive .....	15
3.2.5 Data Mapping for Procedure Arguments .....	15
3.3 Data Parallelism .....	17
3.3.1 FORALL statement .....	17

3.3.2	INDEPENDENT directive . . . . .	17
3.4	Performance Issues . . . . .	18
3.4.1	Sequential Bottlenecks . . . . .	18
3.4.2	Communication Costs . . . . .	19
3.4.3	Limitations of HPF . . . . .	20
<b>4</b>	<b>The Variational Moments Equilibrium Code (VMEC) System . . . . .</b>	<b>22</b>
4.1	VMEC System . . . . .	22
4.2	Space Transform Subroutines . . . . .	24
4.3	Tokamak and Stellarator . . . . .	25
<b>5</b>	<b>Porting VMEC System to Cray T3E . . . . .</b>	<b>27</b>
5.1	Cray T3E . . . . .	27
5.2	Portland Group HPF . . . . .	29
5.2.1	Portland Group HPF . . . . .	29
5.2.2	F90 Features and HPF features Unsupported in PGHPF . . . . .	30
5.3	Problems and Solutions . . . . .	31
<b>6</b>	<b>Partial Parallelization of VMEC System . . . . .</b>	<b>38</b>
6.1	Vector Modifications . . . . .	38
6.1.1	Array Operations . . . . .	38
6.1.2	Matrix Operations . . . . .	39
6.1.3	Result . . . . .	40
6.2	Data Parallelism . . . . .	41
6.2.1	FORALL statement . . . . .	42
6.2.2	INDEPENDENT do loops . . . . .	43
6.3	Data Mapping . . . . .	46
6.3.1	Distributed Arrays . . . . .	46
6.3.2	Nondistributed Data . . . . .	48
<b>7</b>	<b>Testing Result and Analysis . . . . .</b>	<b>51</b>
7.1	2D Tokamak Equilibrium . . . . .	51
7.2	3D QOS Stellarator Equilibrium . . . . .	54
7.3	Conclusion . . . . .	55

**8 Conclusion** ..... 59

**REFERENCES** ..... 61



## ACKNOWLEDGMENTS

I want to thank my advisor, Dr. Ware, for giving me the chance to work on this interesting project. During the whole project, he always showed great help and patience. He has spent a lot time helping me understand the physics background knowledge, analyzing and starting up the project, and finding solutions when problems are found. I also want to thank Dr. Morton, who has provided very helpful advice on deciding some important issues of this project and also on solving several HPF problems.

I want to thank my husband, Yong, for his help and encouragement. Only with his support, could I have concentrated on this project. At last I would like to thank my parents for letting me come to this university to study.

The support of this work by National Energy Research Scientific Computing Center and Arctic Region Supercomputing Center is gratefully acknowledged.

## CHAPTER I INTRODUCTION

Parallel processing is making a tremendous impact on many areas of computer applications. With the high computing power of parallel computers, it is now possible to address many applications that were until recently beyond the capability of conventional computing techniques. Parallel processing is extensively used in areas like weather prediction, biosphere modeling, and pollution monitoring, as well as in scientific computing.

In this project, we tried to port an existing program called Variational Moments Equilibrium Code (VMEC) to parallel structures. VMEC is used in plasma physics to find the equilibrium state of a given plasma. The original version of VMEC was written in 1986 by S. P. Hirshman and the current version has been updated to Fortran 90. Although it can be run conveniently on a variety of different platforms, it usually takes a long time for complicated problems. This is because of the large number of scientific computations in the code and its modular structure. This problem is especially obvious for large input files. To find the equilibrium status of the plasma, a large number of poloidal and toroidal Fourier modes is usually required for a good representation of the equilibrium. And for each Fourier mode, the magnetic field needs to be calculated. In some cases, there can be more than 1000 modes, and the calculations for all those modes will take a long time. In order to make the code run more efficiently for large input files, optimization and parallelization of the code is necessary.

In this project, the VMEC is ported to parallel structure using High Performance Fortran. The parallel version of VMEC is implemented and tested on the Cray T3Es at NERSC (National Energy Research Scientific Computing Center). The HPF compiler used is the Portland Group HPF (PGHPF).

The process of parallelizing VMEC can be divided into two steps. First, modifications are made to optimize the sequential performance and the code structure of VMEC. Then, two space transform subroutines are parallelized with the data mapping and parallelism features in HPF.

In the following chapters, chapter 2 will give a brief introduction to parallel processing, and chapter 3 will discuss the High Performance Fortran language. Chapter 4 is an introduction to VMEC and some of the background knowledge in plasma physics. Chapter 5 and chapter 6 are detailed descriptions of the two steps in parallelizing VMEC, as mentioned in the above paragraph. The parallel code is tested for two kinds of plasma configurations: tokamak and stellarator. The timing results and the analysis is given in chapter 7. Finally, chapter 8 talks about conclusions and future work.

## CHAPTER II PARALLEL PROCESSING

This chapter is a brief introduction to parallel processing. The following sections will talk about some of the most popular parallel computer architectures and parallel programming paradigms, as well as some concepts of parallel processing.

### 2.1 Parallel Computers

A *parallel computer* is a set of processors that are able to work cooperatively to solve a computational problem. We usually call the traditional sequential computer architecture SISD (single instruction single data). For parallel computers, some of the most important architectures are: SIMD (single instruction multiple data), MIMD (multiple instructions multiple data), and multicomputers.

SIMD machines take advantage of the fact that a lot of programs apply the same operation to many different data sets in succession. In a SIMD machine, all processors execute the same instruction stream on a different piece of data. This approach has less complexity for both hardware and software compared to other parallel architectures but is appropriate only for specialized problems characterized by a high degree of regularity, for example, image processing and certain numerical simulations.

For a broader range of parallel programs, such as programs that need each processor to execute a separate instruction stream and work on different data, there are MIMD computers. MIMD computers are probably the most popular supercomputer

architecture today because of their flexibility, and because manufacturers can take advantage of economies of scale by building such machines with hundreds or thousands of standard, and relatively cheap microprocessors. Unfortunately, greater flexibility also makes MIMD computers more difficult to program than the SIMD architectures.

The *multicomputer* is in many ways very similar to *distributed-memory* MIMD computers. It comprises a number of computers linked by an interconnection network. Each computer executes its own program on its own data set. The principal difference between a multicomputer and the distributed-memory MIMD computer is that in a multicomputer, the cost of sending a message between two nodes is independent both of node location and other network traffic, while in the distributed-memory MIMD it is not[7].

Two classes of computer systems that are sometimes used as parallel computers are the local area network (LAN), and the wide area network (WAN). In a LAN system, computers in close physical proximity are connected by a fast network while in a WAN system geographically distributed computers are connected. Ethernet and asynchronous transfer mode (ATM) are commonly used network technologies in such systems[4].

## 2.2 Parallel Computations

A parallel computer is of little use without efficient parallel algorithms. The issues involved designing parallel algorithms are very different from those involved designing their sequential counterparts. A significant amount of work is being done

to develop efficient parallel algorithms for a variety of parallel architectures. Some of the most important parallel programming paradigms include: data parallelism, message passing and shared memory.

### 2.2.1 Data Parallelism

Data parallelism exploits the fact that many programs apply the same operation to each element of a composite data structure, such as an array or list, before applying any other operation to any other data structure. So if we can apply data decomposition to the data structure, the operations on different portions of the data can be carried out concurrently.

The main advantage of the data parallel programming model is that it makes programs easier to write and to read. The main drawback of the data-parallel model is that it is hard to express irregular or heterogeneous computations in it. Algorithmic decomposition, for example, cannot be implemented, since a pipeline's different stages usually need to execute different operations at the same time. Similarly, as the computations to be carried out on the elements of a composite data structure become more dependent on the values of those elements, or their past histories, data parallelism becomes less helpful.

### 2.2.2 Shared Memory

In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously. Various mechanisms such as locks and

semaphores may be used to control access to the shared memory. An advantage of this model is that it simplifies the program development since there is no need to specify explicitly the communication of data from producers to consumers due to the lack of data “ownership”. However, it also makes programming more difficult because of the difficulties in understanding and managing locality in such models.

### 2.2.3 Message Passing

Message passing is the main alternative to shared-memory programming models on present-day parallel computers and it is probably the most widely used parallel programming model today. In a message passing program, processes do not communicate through shared data structures; instead, they send and receive discrete messages to and from named tasks. Message passing programs create multiple tasks, with each task encapsulating local data. The main advantage of message passing model over shared memory model is modularity: by eliminating shared structures, and making both the reading and writing ends of communication explicit, the software can be more robust[7]. Also, by enabling the programmer to handle communication details, programming in message passing is more flexible. However, the cost of these advantages is that programming becomes more complicated and the programs are more error-prone.

### 2.3 Performance Issues

There are two major components of parallel algorithm design. The first one is the identification and specification of the overall problem as a set of tasks that can be performed concurrently. The second is the mapping of these tasks onto different processors so that the overall communication overhead is minimized[14]. The first component specifies concurrency, and the second one specifies data locality. The performance of an algorithm on a parallel architecture depends on both. Concurrency, also called parallelism, is necessary to keep the processors busy. Locality is important because it determines communication cost. Ideally, a parallel algorithm should have maximum concurrency and locality. However, for most algorithms, there is a trade-off. An algorithm that has more concurrency often has less locality.



## CHAPTER III HIGH PERFORMANCE FORTRAN

High Performance Fortran (HPF) is an extended version of Fortran 90 for parallel computer systems. It combines the full Fortran 90 language with special user annotations dealing with data distribution. The new features provided by HPF include: mapping data to multi-processors, specifying data parallel operations and methods for interfacing HPF programs to other programming paradigms[6]. This chapter will give a brief description of some of those features in HPF and how to implement those features.

### 3.1 Basics of High Performance Fortran

For most parallel programming languages, it is up to the programmer to handle all the details of parallelism as well as the communications between processes, which, as a result, will put a very extensive knowledge requirement and intensive amount of work on the programmer. Compared with those parallel programming languages, HPF uses a very high-level data mapping strategy to load much of the burden from the programmer to the compiler. The user of HPF needs to give the compiler information about the program and the data mapping strategy the user intended. The system will generate the details of the communication according to the data mapping strategy and the information of the program the user implied. However, it is still in great part the programmer's responsibility to minimize the communication cost when deciding the data mapping pattern.

### 3.1.1 Fortran 90

Since Fortran 90 is the basis for HPF, we will give a brief introduction to the main features of Fortran 90, especially those that have an impact on HPF.

Fortran 90 (F90) is a complex language. It augments Fortran 77 with pointers, user-defined datatypes, modules, recursive subroutines, dynamic storage allocation, array operation, new intrinsic functions, improved input and output, and many other features. Among all the new features, two of them are most relevant to parallel programming: the array assignment statement and the array intrinsic functions[6]. We will here focus on these two features.

The array assignment statements in Fortran 90 allow operations on entire arrays without explicit DO loops. Following is an example of how a nested do-loop in Fortran 77 can be expressed in one simple array assignment statement in Fortran 90:

```
Fortran 77:  DO i = 0, 10
              DO j = 0, 10
                A(i,j) = B(i,j) + C(i,j)
              END DO
            END DO
```

```
Fortran 90:  A = B + C.
```

The array assignment statement in Fortran 90 provides for element-by-element operations on entire arrays. When executing such a statement, the compiler will make sure that the entire right-hand side of an assignment is evaluated before the left-hand

side is modified, and prohibit attempts to do multiple updates to a left-hand side. In doing so, the particular order of evaluation is not specified by the language. Such semantics of Fortran 90 allow these array assignment statements to be executed in parallel. For example, in HPF, if the arrays associated with the left-hand-side of the expression are distributed over processors, then each node or processor on the parallel system will execute only its local part of the computation.

All Fortran intrinsic functions that apply to scalar values can also be applied to arrays, in which case the function is applied to each array element. And, when the array elements are distributed over processors in a parallel architecture, just as with the array assignment statements, the intrinsic function can also be parallelized by localizing array indices. Some of the array intrinsic functions provided by Fortran 90 include: MAXVAL, MINVAL, SUM, PRODUCT, MAXLOC, MINLOC, MATMUL, DOT\_PRODUCT, TRANSPOSE and CSHIFT[4].

### 3.1.2 Compiler Directives

Both array assignment statements and array intrinsic functions are explicit parallel operations that the compiler can detect easily. For those parallel structures that are hard to detect, HPF provides compiler directives for the programmer to suggest implementation strategies or assert facts about a program to the compiler. Compiler directives help the compiler to detect as much parallelism in the program as possible.

Compiler directives form the heart of the HPF language. Directives are actually only Fortran comments. Thus, they may be ignored by a standard Fortran compiler.

But, to an HPF compiler, although most directives are not directly executable, they can supply the information needed to optimize the performance, while not changing the value computed by the program. A HPF directive has one of the following forms:

`!HPF$ hpf-directive`

`CHPF$ hpf-directive`

`*HPF$ hpf-directive`

The first form above is the most recommended because it is the only form that works for free source form in Fortran 90 syntax[6]. Most of the parallelism features in HPF are expressed as compiler directives.

### 3.1.3 Parallelism Features

In HPF, the two most important parallelism features -and probably the most publicized features- are data mapping and data parallelism.

Data mapping describes how data is divide among the processors in a parallel machine. It implicitly determines the communication patterns in a program. In HPF, there are two data-to-processor mapping stages: the DISTRIBUTION and ALIGN directives.

Data parallelism describes operations in the program that can be performed in parallel if the computer has the resources. There are two main data parallel constructs in HPF: the FORALL statement and the INDEPENDENT directive.

Besides data mapping and data parallelism features, HPF also provides a large set of intrinsic functions and library procedures. Many of them are data parallel

operations. The user can also get information about the state of the machine or an array's distribution using a number of inquiry subroutines in HPF. The rest part of this chapter will describe some features in HPF that were used in this project.

### 3.2 Data Mapping

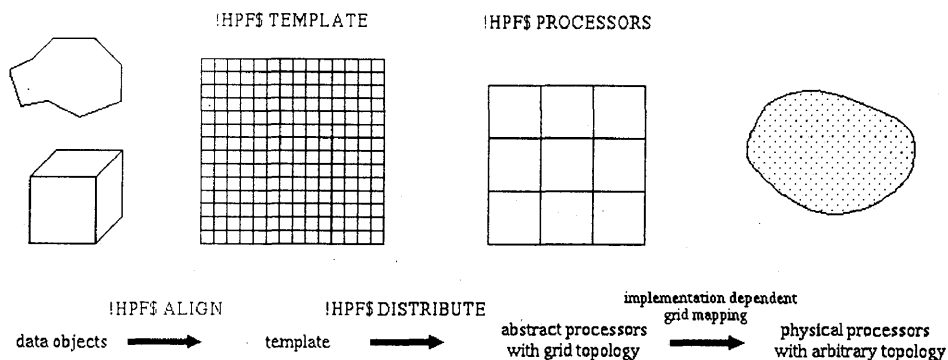


Figure 3.1:

Data mapping in HPF is described in Figure 3.1 as a three-level model: first, arrays are aligned relative to one another using `ALIGN` directives; then, this group of arrays is distributed onto a user-defined, rectilinear arrangement of abstract processors using `DISTRIBUTE` and `PROCESSORS` directives; the final mapping from abstract to physical processors is not specified by HPF and it is language-processor dependent.

### 3.2.1 DISTRIBUTE directive

The DISTRIBUTE directive specifies a mapping of data objects to abstract processors in a processor arrangement. Technically, the distribution step of the HPF model applies to the template of the object to which the array is ultimately aligned.

Each dimension of an array may be distributed in one of three ways:

- \* No distribution
- BLOCK(n) Block distribution (default:  $n=N/P$ )
- CYCLIC(n) Cyclic distribution (default:  $n=1$ )

Some examples are illustrated in Figure 3.2.

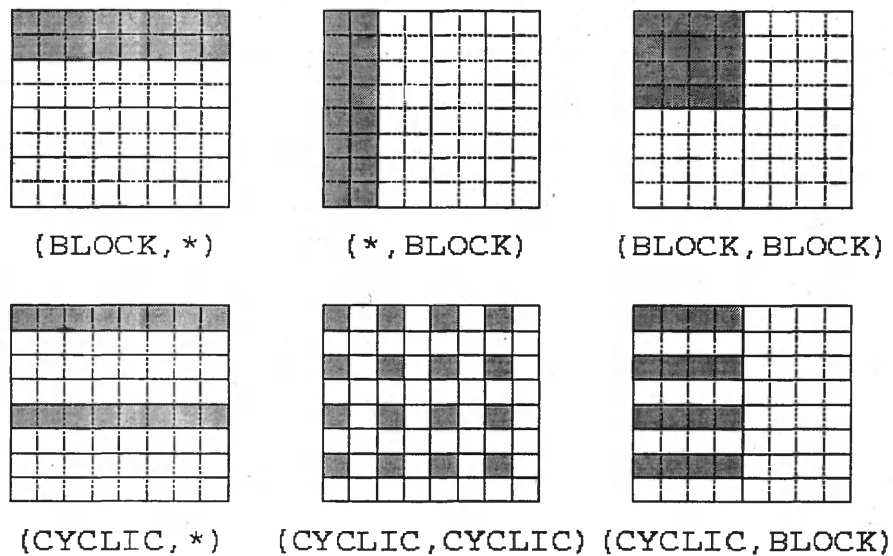


Figure 3.2:

### 3.2.2 ALIGN directive

The ALIGN directive is used to specify that certain data objects are to be mapped in the same way as certain other data objects. Operations between aligned data

objects are likely to be more efficient than operations between data objects that are not known to be aligned. Examples of ALIGN statements are shown in Figure 3.3.

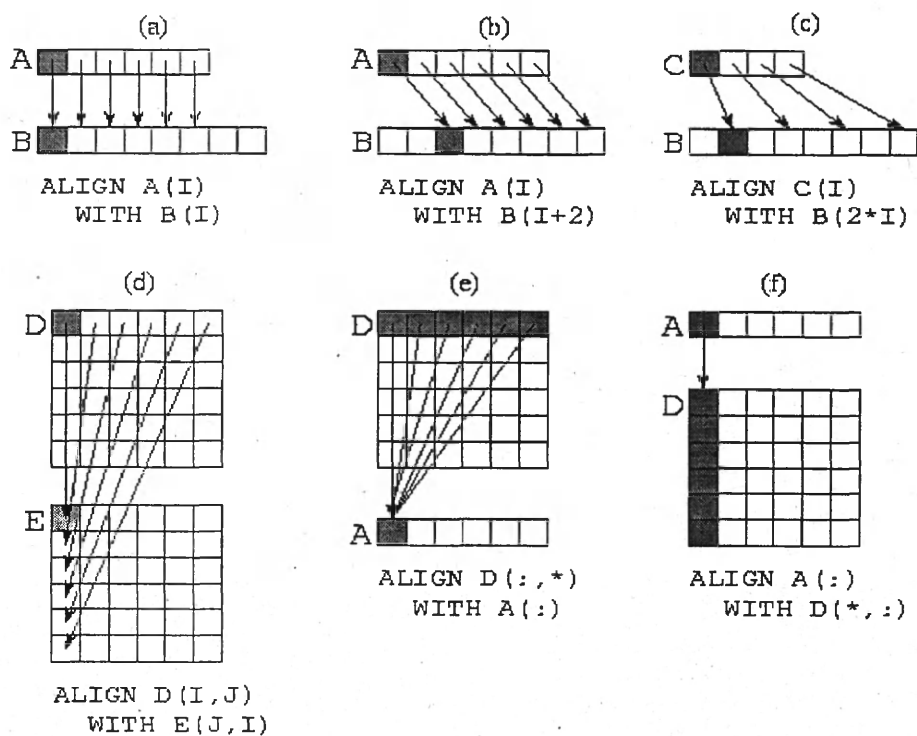


Figure 3.3:

Note that it is illegal to explicitly realign an object if anything else is aligned to it and it is illegal to explicitly redistribute an object if it is aligned with another object.

### 3.2.3 TEMPLATE directive

The `TEMPLATE` directive declares one or more templates of a certain rank and shape each time the data is distributed. In HPF, we can think of each array as being aligned with a specific template. If no template is explicitly declared for an array, by default, it is aligned to its natural template, i.e. template with the same rank and

shape as the array. The following are some examples of `TEMPLATE` directives:

**Example 1** *Examples of `TEMPLATE` directives:*

```
!HPF$ TEMPLATE T1(100), T2(N,2*N)
```

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK) :: A(N)
```

### 3.2.4 PROCESSOR directive

The `PROCESSOR` directive declares one or more rectilinear processor arrangements with specific rank and shape[4]. Only rectilinear processor arrangements are allowed in HPF.

**Example 2** *Examples of `PROCESSORS` directives:*

```
!HPF$ PROCESSORS P(N)
```

```
!HPF$ PROCESSORS BIZARRO(1972;1997, -20:17)
```

The final mapping of abstract to physical processors is not specified by HPF, and it is language-processor dependent. However, if two objects are mapped to the same abstract processor at a given instance during the program execution, the two objects are mapped to the same physical processor at that instant.

### 3.2.5 Data Mapping for Procedure Arguments

Since the actual argument and the dummy argument has separate templates, they don't necessarily have to be mapped the same way. So, when calling subroutines,



we often face one of the following situations related to the mapping of the dummy arguments:

1. The mapping of the dummy arguments is known at compile time and it is to be enforced regardless of the mapping of the actual argument. In this case, the mapping of the dummy argument must be defined explicitly, and it must also appear in interface blocks.
2. The mapping of the dummy argument is known at compile time and it is the same as that of the actual argument. In this case, we use a descriptive form of mapping directives with asterisks preceding the mapping specifications.

**Example 3** *Descriptive mapping of the dummy argument:*

```
!HPF$ DISTRIBUTE A *(BLOCK)
```

The above example asserts the compiler that A is already distributed BLOCK onto processors so, if possible, no data movement should occur.

3. The mapping of the dummy argument is not known at compile time and it should be the same as that of the actual argument. In this case, we use a transcriptive format of mapping directives.

**Example 4** *Transcriptie mapping of the dummy argument*

```
!HPF$ DISTRIBUTE A * ONTO *
```

The above example specifies that mapping of A should not be changed from that of the actual argument.

### 3.3 Data Parallelism

The HPF language in conjunction with Fortran 90 array features provides several methods for the programmer to convey parallelism which the HPF compiler will detect and parallelize. This section describes the FORALL statement and the INDEPENDENT directive.

#### 3.3.1 FORALL statement

The FORALL statement provides a convenient syntax for simultaneous assignments to large groups of array elements. The functionality they provide is very similar to that provided by the array assignments.

**Example 5** *FORALL statement:*

$$\text{FORALL}(I=1:100) B(I) = 1.0$$

In FORALL blocks, the array elements may be assigned in an arbitrary order, in particular, concurrently. Each array element must be assigned only once to preserve the determinism of the result.

#### 3.3.2 INDEPENDENT directive

The INDEPENDENT directive asserts that the iterations of a DO or FORALL do not interface with each other in any way. By preceding a DO loop or a FORALL statement, the directive provides information about the program the compiler will use to parallelize and optimize the execution of the program. For example:

```
!HPF$ INDEPENDENT
```

```
FORALL (I=1:100) B(I)=1.0
```

### 3.4 Performance Issues

Since HPF is a very high level parallel programming language, the performance of a program depends not only on the skill of the programmer but also on the capability of the compiler.

There are two major obstacles that impact the performance of an HPF program: sequential bottlenecks and excessive communication costs. In the following subsections, we will discuss these two obstacles.

#### 3.4.1 Sequential Bottlenecks

A sequential bottleneck occurs when a code fragment is not parallelized sufficiently or when parallelism exists but cannot be detected by the compiler. In either case, the code fragment can only be executed sequentially. In situations where the program is relatively small and is only going to execute on a small number of processors, the sequential bottleneck may be insignificant. But for large programs, and especially for those intended to run on a large number of processors, this bottleneck can have great impact on the effectiveness of parallelism. According to Amdahl's law, if some fraction  $1/s$  of a program's total execution time executes sequentially, then the maximum possible speedup that can be achieved on a parallel computer is  $s$ . Thus, the smaller the fraction of code that executes sequentially, the greater speedup we can get.

### 3.4.2 Communication Costs

There are actually several issues that can affect the communication cost of HPF programs. The first one is array assignments. Array assignments and FORALL statements can result in communication if the computation on one processor requires data values from another processor. Also, cyclic distributions will often result in more communication than will block distributions. However, by scattering the computational grid over available processors, better load balance can result in some applications.

Different mappings of arrays is another main source of communication cost. Any operation performed on nonaligned arrays can result in communication. But, to convert the arrays to a common distribution before the operation will cause another kind of communication cost, array remapping.. So, extra precautions should be made for this kind of problem.

Procedure boundaries will often cause communication costs, too. This kind of communication often occurs when the distribution of the dummy arguments differs from the distribution of the actual arguments, since, for each subroutine, there is often a distribution of its dummy arguments and local variables that is optimal in the sense that it minimizes execution time in that subroutine. However, this optimal distribution may not correspond to the distribution specified in the calling program. This will result in the different distributions for the actual arguments and the dummy arguments, which may cause high communication costs when remapping the array from actual arguments to the dummy arguments when the subroutine is called, then

later from the dummy arguments back to the actual arguments when the subroutine returns. To reduce such communication cost, we need to evaluate different data mapping approaches carefully and choose the optimal data mapping strategy considering the whole structure of the program.

### 3.4.3 Limitations of HPF

Compared to other popular parallel programming languages and tools like MPI and PVM, programmers for HPF are freed from the job of generating communication code and can focus on the tasks of identifying opportunities for concurrent execution and determining efficient partition, agglomeration, and mapping strategies. However, since the communication cost of a program is directly determined by its data mapping strategy, it is still the programmer's responsibility to choose the optimal data mapping for the program to minimize the overhead in communication.

Another limitation of HPF is the limited range of parallel algorithms that can be expressed in HPF. With the compiler directives and other parallel features, HPF can only be targeted to the SPMD programming model. Thus, its effectiveness is limited to programs that are suitable for data decomposition or programs that contain intensive array operations. For programs with large portions of serial code embedded in them, the usage of HPF may cause very high overhead cost and is not recommended.

Finally, although a HPF DO loop can be executed using INDEPENDENT directives, there is no way to express the inter-dependence of statements within a DO loop. Therefore, all statements in the DO loop under the same loop index have to be

executed serially. This also limits the full parallelization of the code.

# CHAPTER IV

## THE VARIATIONAL MOMENTS EQUILIBRIUM CODE (VMEC) SYSTEM

### 4.1 VMEC System

Plasma is currently an active research area in the physics society. The practical terrestrial applications of man-made plasmas are very extensive. They range from the microfabrication of electronic components to demonstrations of substantial thermonuclear fusion power from magnetically confined plasmas. In studying plasma, the concept of magnetohydrodynamic (MHD) is often used. MHD provides a macroscopic dynamical description of an electrically conducting fluid in the presence of magnetic fields. MHD has been very successful in solving problems in plasma, such as: finding magnetic field configurations capable of confining a plasma in equilibrium, the linear stability properties of such equilibria and the nonlinear development of instabilities and their consequences[3].

The basis of this project is an existing program called VMEC (Variational Moments Equilibrium Code), which solves three-dimensional MHD equilibrium equations using Fourier Spectral (Moments) Methods.

VMEC consists of two parts. The first part of the program is the equilibrium solver. It calculates the equilibrium state of a given plasma by minimizing the total energy - magnetic plus thermal - of a plasma confined in a toroidal domain  $\Omega_p$ :

$$W_p = \int_{\Omega_p} \left( \frac{1}{2} B^2 + p \right) dV$$

To calculate the magnetic field of the plasma, both a cylindrical coordinate representation ( $R, Z, \Phi$  coordinates) and a magnetic coordinate representation ( $s, \zeta, \theta$  coordinates) are used. In the magnetic coordinate,  $s$  is the flux surface label, which is equal to 1 on the outermost surface of the plasma and is 0 for the innermost surface, i.e. the poloidal axis of the plasma.  $s$  is proportional to  $r^2$ , in which  $r$  is the *radial coordinate* (as shown in Figure 4.1(b)). In the magnetic coordinate, the calculation is carried out by dividing the toroidal domain of the given plasma into different surfaces along *radial*( $r$ ) *coordinate*, then each surface is further divided into small areas by grid points along *poloidal*( $\theta$ ) and *toroidal*( $\zeta$ ) *coordinate*. On each surface, the plasma pressure remains constant in equilibrium state[12].

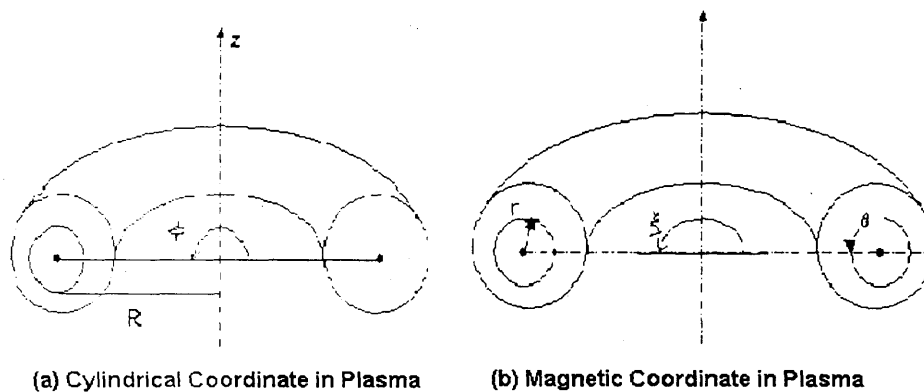


Figure 4.1:

The second part of the program is the optimizer. In this part, several target parameters are defined. After each equilibrium of a plasma is solved, VMEC calculates its “distance” from the target plasma. Then changes the input parameters and checks



to see if it has moved closer to the target. This process is carried out by calling the equilibrium code repeatedly to find the nearest solution to the target plasma. The most recent version of VMEC optimizer contains a fast ballooning code(COBRA) to include ballooning stability in the optimization.

The original VMEC was written in Fortran 77 by S. P. Hirshman in 1985. New features have been added to the code constantly since then and the code has been updated to Fortran 90. The current code is version 5.20, which is also the version used in this project.

## 4.2 Space Transform Subroutines

In this project, we targeted on parallelizing the VMEC equilibrium solver, which contains the major calculations in the whole program. There are about 40 subroutines in the this part of the code. Among them, two subroutines contribute to almost 40% of the whole equilibrium calculation time. Therefore, we focused our efforts first on parallelizing these two subroutines.

These two subroutines are called space transform subroutines. What they do is to transfer from real space to Fourier space before equilibrium calculation and transfer MHD forces from Fourier space back to real space after the calculation is done. Computations performed in these two subroutines are:

$$B(\theta, \zeta) = \sum_m \sum_n B_{m,n} \cos(m\theta - n\zeta)$$

$$B_{m,n} = \frac{1}{2\pi} \int \int B(\theta, \zeta) \cos(m\theta - n\zeta) d\theta d\zeta$$

The number of calculations in the above equations depends on the maximum values of  $m, n, d\theta$  and  $d\zeta$ , which are determined by the values in the input file. For equilibriums with a lot of structure, more than 10,000 grid points can be used in the calculation.

### 4.3 Tokamak and Stellarator

Two kinds of plasma configurations are used in this project to test the performance of the parallel version VMEC: the tokamak and the stellarator.

The tokamak is a toroidally symmetric plasma trap that uses a large plasma current to produce a confining poloidal magnetic field[5]. Because of its symmetry along the toroidal coordinate, we only need to consider the magnetic field along the other two coordinates, radial coordinate and poloidal coordinate. Therefore we can think of the tokamak as a 2D equilibrium and it requires much less calculation than a 3D stellarator (explained in the following paragraph). The tokamak input file used in this project has 558 Fourier modes and the magnetic field is calculated for each of the Fourier modes.

Stellarators are nonsymmetric plasma traps relying on external coils to produce the internal transform needed for the confinement and stability[5]. Since there is no symmetry along any of the magnetic coordinates, all three coordinates need to be considered when the magnetic field is calculated. This usually results in far more

Fourier modes in the plasma and heavier computation load for the program than a 2D tokamak. The stellarator input file used in this project contains 11,016 Fourier modes.

## CHAPTER V PORTING VMEC SYSTEM TO CRAY T3E

The parallel computer system we have chosen to port the VMEC system to is the Cray T3E computer system at the National Energy Research Scientific Computing Center (NERSC) located at Lawrence Berkeley National Laboratory. The HPF compiler we used is Portland Group HPF (PGHPF).

The serial version of VMEC is written in Fortran 90 and has never been tested in the Cray T3E. Therefore, before we parallelize VMEC, modifications had to be made to the program to make it run smoothly on the Cray T3E machine, and for the PGHPF compiler.

This chapter will first give a brief introduction of the Cray T3E machine and the PGHPF compiler, and then the detailed description of changes made to VMEC in this first phase of the project.

### 5.1 Cray T3E

The Cray T3E machine used in this project is named *mcurie*. It is one of the six high-performance Cray research computer systems at NERSC. *Mcurie* is a distributed-memory “Massively Parallel Processor” (MPP) computer with 695 individual processors, each one capable of performing 900 million floating point operations per second (MFLOPS). All processors and disks are connected via a custom high speed network.

The processors on the Cray T3E are manufactured by Digital Equipment Corpo-

ration (DEC), and are known as Alpha chips. The Alpha chips have a clock speed of 450 MHz, and can perform one floating point add and multiply per clock cycle, giving each PE a theoretical peak speed of 900 million floating point operations per second (MFLOPS).

In the Cray T3E, each processor has its own local memory. Together with some network interface hardware, the processor and local memory form a Processing Element (PE). The PEs are connected by a network arranged in a 3-dimensional torus. And in the torus, each PE is considered topologically equivalent - the concept of "near neighbors" is not useful on the T3E as it might be on other distributed-memory parallel computers.

Each PE of the Cray T3E has a 256 MB of memory that it can address directly. The operating system uses approximately 12 MB on each PE, leaving about 244 MB available for user code. The content of memory on other PEs is available by passing messages via subroutine calls defined in message passing libraries (known as PVM, MPI and SHMEM), or by using the data-parallel programming language HPF.

Among the 695 PEs of the Cray T3E machine, there are 640 application (APP) PEs. These are the PEs that run parallel jobs. The other PEs, known as command (CMD) PEs and operating system (OS) PEs, run single-processor user commands and perform system functions, respectively. For example, when the users log into mcurie interactively using telnet, they are running on a CMD PE.

The operating system for Cray T3E is called UNICOS/mk(microkernel). It is designed to replace regular UNIX by serverizing it into smaller, more manageable

components. It provides features like: basic hardware abstraction, memory management, CPU scheduling, thread scheduling and inter-processor communication(IPC).

The Cray T3E programming environment supports programming in Fortran 90, High Performance Fortran, C, C++ and assembler.

The Cray T3E also supplies tools to help the user debug and analyze MPP programs. The debugger on Cray is called “totalview.” TotalView is a source-level debugger and can be used to debug C, C++, High-Performance Fortran (HPF), and Fortran 90 programs. Another useful tool on Cray T3E is called “apprentice”. It is a performance analysis tool that helps the user find and correct performance problems and inefficiencies in programs. It can work with C++, Cray Standard C, Fortran 90 and PGHPF compilers. These tools and other performance analysis tools (PAT) on Cray T3E provides a low-overhead method for estimating the amount of time spent in functions, determining load balance across processing elements (PEs), generating and viewing trace files, timing individual calls to routines, performing event traces, and displaying hardware performance counter information.

## **5.2 Portland Group HPF**

### **5.2.1 Portland Group HPF**

The HPF language used in this project is the Portland Group’s implementation of HPF version 2.4. This version conforms to the High Performance Fortran Language Specification Version 1.1, published by the Center for Research on Parallel Compu-

tation, at Rice University, with a few limitations and modifications to the standard High Performance Fortran Language Specifications.

Components provided in PGHPF 2.4 include: PGHPF High Performance Fortran Compiler, the PGPROF graphical profiler and the support for the TotalView multiprocess debugger. PGHPF 2.4 is supported on a variety of High Performance Computers, workstations and clusters. In particular, some of the supported systems include: LINUX, Cray T3E (UNICOS/mk2.0,2.25), Cray J90, Cray C90, Cray T90, IBM RS6000/SP (SP2), IBM RS6000 workstations running AIX 4.x and Intel Paragon (cross compilers on SPARC systems running Solaris 2.4 or higher).

### **5.2.2 F90 Features and HPF features Unsupported in PGHPF**

Although PGHPF is declared to be a superset of Fortran 90 and conforms with the standard HPF language specification, there are some restrictions to the Fortran 90 and HPF features supported in PGHPF. This caused some problems when porting VMEC to HPF. Following are some of these restrictions.

*Fortran 90 pointer restrictions.* In PGHPF2.4, pointers cannot be in COMMON blocks and they can appear in a module only if they are not distributed; pointers cannot be DYNAMIC; a scalar pointer cannot be associated with a distributed array element; a TARGET object cannot have CYCLIC distributions; and a pointer dummy variable cannot be used to declare other variables.

*Module restrictions.* Named array constants defined in a module cannot be used as an initializer in a subprogram which USES the module; named array or structure

constants found in modules cannot be used in either of the following: values in CASE statements, kind parameters in declaration statements, kind argument in intrinsics or initial values in parameter statements or declaration statements.

*DISTRIBUTE and ALIGN restrictions.* PGHPF 2.4 ignores the distribution directives applied to character types, arrays subject to a SEQUENCE directive, and NAMELIST arrays.

Besides the above restrictions, there are also unsupported features in Fortran 90 – derived types, named constants, optional argument, PURE statement and HPF\_LIBRARY routines. Since those restrictions do not have much impact on this project, we will omit their details.

### 5.3 Problems and Solutions

The original VMEC code contains Unix script commands in it. It uses the C-precompiler to produce both the machine-specific Fortran source code and makefiles. The Cray T3E were not in its list of platforms. Therefore, options for the Cray T3E were added to the script so that the Fortran code and makefiles will take up the correct function names and compiler options.

When porting VMEC to PGHPF, there were more modifications made to the code because of the unsupported Fortran 90 features in PGHPF 2.4. Changes made to the code in this phase include:

1. *Namelists in the modules:* The PGHPF compiler doesn't allow more than one module that contains namelists to be used in another module or a subroutine. For such a



situation, the compiler will give an error message on “unrecognized symbol.” To resolve this problem, we moved the namelists from the modules to all the corresponding subroutines.

2. *Allocatable character arrays*: The PGHPF compiler can not recognize allocatable character arrays. For this problem, we changed all the allocatable character arrays to be nonallocatable.

3. *Argument passing*: The Fortran 90 version of VMEC used a lot of subroutine calls in which the actual arguments had different ranks and shapes than the dummy arguments ( as shown in Example6). This is allowed in Fortran 90 because of sequence association (the order of array elements that Fortran 90 requires when an array, array expression, or array element is associated with a dummy array argument). Sequence association is a natural concept only in systems with a linearly addressed memory. It is based on the traditional single address space, single memory unit architecture. This model can cause severe inefficiencies on architectures where storage for variables is mapped. As a result, HPF modified Fortran 90 sequence associations rules. In HPF, a distributed array can be passed to a subprogram only if actual and dummy arguments are conformable (they have the same shape). Otherwise both actual and dummy arguments must be declared sequential. If the HPF compiler detects that the actual arguments and the dummy arguments have different shapes for a subroutine call, it will give error messages and abort.

To solve this problem, we made several attempts from different approaches. At first, we tried to declare both the actual arguments and the dummy arguments se-

quentially by inserting SEQUENCE directives (shown in Example7 as solution1). The program worked fine on one processor. However, for multiprocessors, the distribution of the sequential arrays are ignored by the compiler. This is because of the PGHPF compiler's restriction on distributing sequential variables, as we mentioned in the previous section. Since the data mapping failed, the program can not run in parallel.

Another solution to this problem is suggested by using the RESHAPE function (as shown in the Example8 as solution2)[6]. But, we later found out that the PGHPF compiler worked differently from the what the standard HPF language specification suggests. In the called subroutine, if the dummy argument's value is changed, the corresponding actual argument will not reflect the changes after the called subroutine returns. This caused the result to be incorrect.

We modified solution2 to be solution3 in Example9. Solution3 uses one RESHAPE function both before and after the subroutine call. Before the subroutine call, RESHAPE is used to map the actual argument to shape of the dummy argument, and the result is stored in a temporary array. This temporary array is then passed to the dummy argument during the subroutine call. After the called subroutine returns, RESHAPE function is used again to copy the elements in the temporary array back to the actual argument so that changes to the dummy argument will show up in the actual argument. This solution works fine on both one processor and multiprocessors. However, this solution caused a new problem: by using a lot RESHAPE functions to copy elements between arrays back and forth frequently, the program is slowed down dramatically.

Finally, we found the optimal solution by combining solution1 and solution3, i.e. solution4 in the Example10. In this solution, we kept all the SEQUENCE directives in solution1, except for those subroutines in which the dummy arguments are going to be mapped across processors. For these subroutines, we used temporary arrays described in solution3. By doing this, we can keep the overhead cost relatively low by using as few as RESHAPE functions as possible while still being able to distribute the dummy arguments where it is needed.

**Example 6** *Fortran 90:*

```

program    ! the calling program

real(kind=rprec), dimension(27) :: a    ! actual argument

call callee(a)

end

subroutine callee(b)

real(kind=rprec), dimension(3,3,3) :: b    ! dummy argument

! actions in subroutine

end subroutine

```

**Example 7** *Solution1:*

```

program    ! the calling program

real(kind=rprec), dimension(27) :: a    ! actual argument

!HPF$ SEQUENCE :: a    ! declare a to be sequential

call callee(a)

```

```

end

subroutine callee(b)

real(kind=rprec), dimension(3,3,3) :: b ! dummy argument

!HPF$ SEQUENCE :: b ! declare b to be sequential

! actions in subroutine

end subroutine

```

**Example 8 Solution2:**

```

program ! the calling program

real(kind=rprec), dimension(27) :: a ! actual argument

call callee(RESHAPE(a, (/3,3,3/)))

end

subroutine callee(b)

real(kind=rprec), dimension(3,3,3) :: b

! actions in subroutine

end subroutine

```

**Example 9 Solution3:**

```

program ! the calling program

real(kind=rprec), dimension(27) :: a ! actual argument

real(kind=rprec), dimension(3,3,3) :: temporaryArray ! temporary array

temporaryArray=RESHAPE(a, (/3,3,3/))

```

```

    call callee(temporaryArray)           ! pass temporary array to dummy argu-
ment
end

subroutine callee(b)

real(kind=rprec), dimension(3,3,3) :: b  ! dummy argument

! actions in subroutine

end subroutine

```

**Example 10 Solution4:**

```

program    ! the calling program

real(kind=rprec), dimension(27) :: a    ! actual argument

!HPF$ SEQUENCE :: a

real(kind=rprec), dimension(3,3,3) :: temporaryArray  ! temporary array

! dummy argument will not be distributed in callee1

call callee1(a)

temporaryArray=RESHAPE(a, (/3,3,3/))

! dummy argument will be distributed in callee2

call callee2(temporaryArray)

end

subroutine callee1(b)

real(kind=rprec), dimension(3,3,3) :: b

! actions in subroutine, b will not be distributed in the subroutine

```

*end subroutine*

*subroutine callec2(b)*

*real(kind=rprec), dimension(3,3,3) :: b*

*!HPF\$ DISTRIBUTE(block, block, block) :: b*

*! actions in subroutine, b is distributed in the subroutine*

*end subroutine*

## CHAPTER VI

### PARTIAL PARALLELIZATION OF VMEC SYSTEM

When porting VMEC to HPF, we focused on implementing the two of the most important features of HPF, the data mapping and the parallelism. From the data mapping perspective, computational related arrays in the space transform subroutines were aligned to each other and distributed over processors. From the parallelism perspective, potentially parallel structures were determined and the compiler was informed by using compiler directives.

Before the parallelization, VMEC was optimized by using array operations to further improve the timing of the program. We call this procedure vector modifications as opposed to parallel modifications in parallelization. Details of vector modifications are described in the first subsection. The next two subsections will describe the two parallelization issues, data mapping and parallelism, respectively.

#### 6.1 Vector Modifications

##### 6.1.1 Array Operations

The original VMEC is coded with Fortran 90. It uses many new Fortran 90 features such as more natural language syntax, data facilities, modularization facilities and intrinsic procedures. However, it does not take much advantage of Fortran 90's array operation feature, which makes it easier for the compiler to determine which operations may be carried out concurrently. So, the first thing we did before parallelizing the program was to use the *array syntax* of Fortran 90 to replace do loops

and nested do loops in the code.

**Example 11** *Use array operations to replace do loops in Fortran 90.*

*Without Array Syntax:*

```
DO I=1, N
```

```
  A(I) = B(I+1)
```

```
END DO
```

*With Array Syntax:*

```
A(1:N) = B(2:N+1)
```

### 6.1.2 Matrix Operations

Besides array syntax, we also used array intrinsic functions to optimize the program. The VMEC, like most of the programs in scientific computing, contains large amount of matrix operations. These matrix operations usually consume a great part of the total execution time. Therefore, by optimizing the matrix operations, not only the code itself is simplified, but also the performance of the program will improve. In VMEC, many matrix operations are implemented in old Fortran 77 style, rather than in Fortran 90 style. In other words, matrix operations are done in explicit nested do loops rather than using Fortran 90 intrinsic functions. To optimize matrix operations in VMEC, we used both DOT\_PRODUCT and MATMUL intrinsic functions. DOT\_PRODUCT calculates the dot-product of two one dimensional arrays and MATMUL calculates the multiplication of two one or two dimensional arrays.



**Example 12** *Use matrix operation intrinsic functions:*

*Without intrinsic functions:*

*DO I=1, N*

*DO J=1, M*

*A(I) = A(I) + B(I,J)\*C(J)*

*D(I) = D(I) + E(J)\*F(J)*

*END DO*

*END DO*

*With intrinsic functions:*

*A(1:N) = MATMUL(B(1:N, 1:M), C(1:M))*

*D(1:N) = DOT\_PRODUCT(E(1:M), F(1:M))*

### 6.1.3 Result

As a result of the vector modifications to VMEC, in one subroutine, the number of do loop nests is reduced from 5 to 2. Keeping the number of do loop nests down will make the structure of the code clearer, and will make it easier for the programmer to recognize the relationships between arrays. It provided a better foundation for the data mapping and the parallelism steps.

By using MATMUL and DOT\_PRODUCT intrinsics, the timing of the program is improved too. The program's execution time on the T3E's is reduced by about 35%, as shown in the following graph:

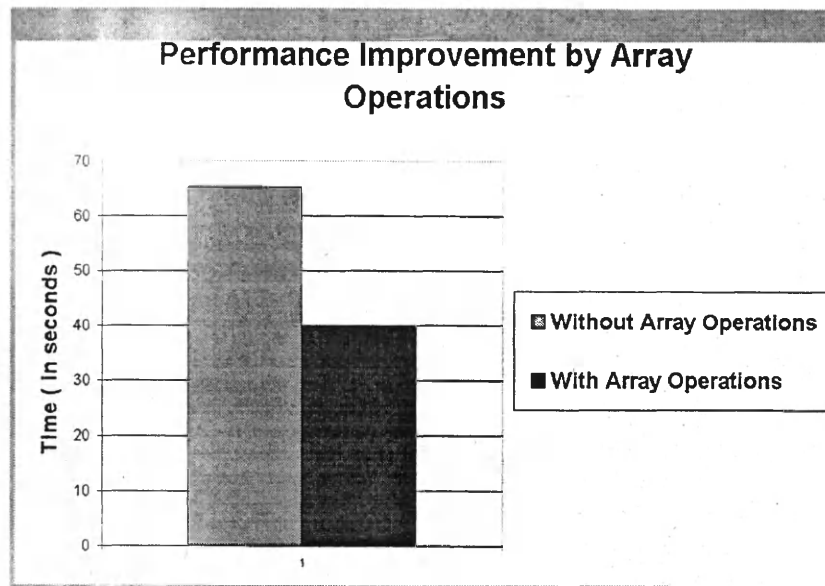


Figure 6.1:

## 6.2 Data Parallelism

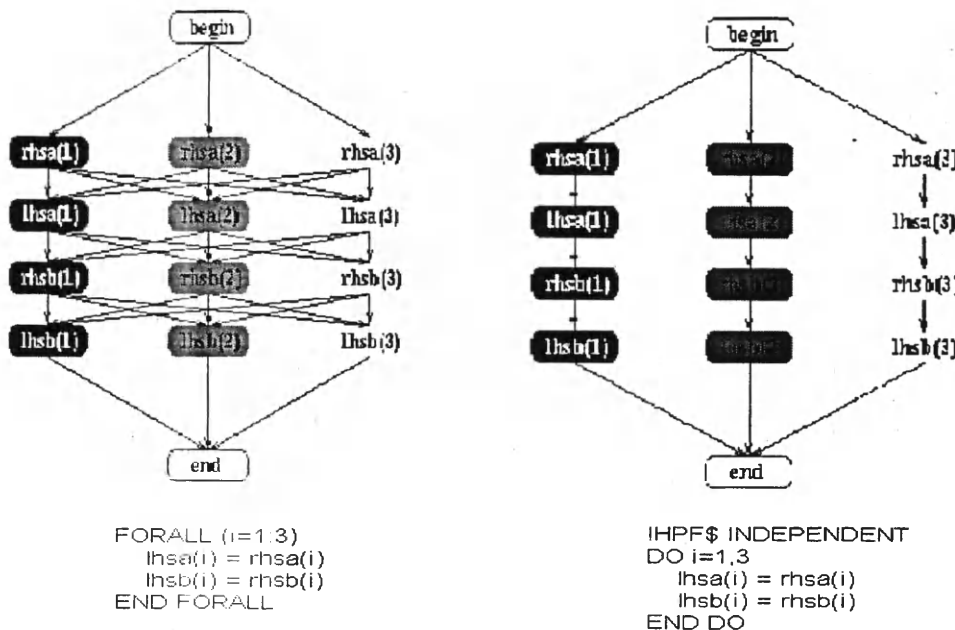
In parallelization, data is usually distributed according to the parallel operations that the data is involved in. Thus data parallelism is usually done before the data mapping phase. Here, we will first explore data parallelism strategies used in this project.

The array syntax we described in the previous section can form implicit parallel operations when the array is mapped across processors. For parallel operations that need to be declared explicitly, `INDEPENDENT` directives are used.

### 6.2.1 FORALL statement

The FORALL statements and INDEPENDENT directives are the two most used parallelism features in HPF. However, we avoided the use of FORALL statements in this project on purpose. There are several reasons for this. The first reason relates to Fortran 90 compatibility. Since the FORALL statement is a new feature in HPF, a Fortran 90 compiler will not recognize it.

The other reason is concurrency. In a FORALL block, the execution of the array assignments may require interstatement synchronizations: the evaluation of the left hand side expression of the FORALL assignment must be completed for all array elements before the actual assignment is made. Then, the processors must be synchronized again, before the next array assignment is processed. In some cases these synchronizations may not be necessary and they can cause longer execution time for the program. Compared to the FORALL statement, each iteration in an INDEPENDENT do loop can be processed independently of any computations performed in other iterations. The diagram and example code in Figure 6.1 illustrate the concurrency for FORALL statements and INDEPENDENT do loops respectively (lines in the diagram symbolize data dependencies).



Although independent FORALL statements are equivalent with the INDEPENDENT do loops in concurrency, we still avoid them because of the compatibility issue. Besides, the user can always use the compiler option to convert the INDEPENDENT do loops to INDEPENDENT FORALL statements during compiling time, if needed.

### 6.2.2 INDEPENDENT do loops

When using INDEPENDENT directives, extra precautions should be given. If the user gives the compiler the wrong information (e.g. assert that a do loop is independent when it is not), and the compiler trusted the information provided by the user, then the do loop will be distributed among processors without question, and the result of the execution will become unpredictable.

For this project, since both of the space transform subroutines have complicated code structures and relationships between arrays, two rules are used to help tell independent loops from dependent loops: *Bernstein's conditions* and the *no control dependence rule*.

Bernstein's conditions says that if  $R_i$  is the "read" operation in iteration  $i$  of a loop, and  $W_i$  is the "write" operation in iteration  $i$ , then for any  $i \neq j$  it must be true that

$$(R_i \cap W_j) \cup (W_i \cap R_j) \cup (W_i \cap W_j) = \emptyset$$

This means that no data object may be read in one iteration and written in another, nor may any data object be written in more than one iteration[6].

The no control dependence rule means that once the construction begins execution, it will execute to completion. These two rules make the task of recognizing independent loops much easier for the programmer and make the result more precise. This is very important for a parallel code since wrong information can lead to incorrect execution result.

However, even if all the independent loops are correctly determined, not all of them can be declared by using INDEPENDENT directives. This is because of the restrictions in PGHPF. PGHPF constrains the maximum number of nested INDEPENDENT loops to be three and there can be at most one INDEPENDENT loop directly nested within another INDEPENDENT loop. In the original VMEC, both of the space transform subroutines contained up to 5 nested do loops and more than one

possible independent loop directly nested within another possible independent loop. So it is very important to use the array syntax and intrinsics to simplify the code first (as described in the previous section). The simplified code still contains three nested do loops and two directly nested within another independent loop, as shown in the following example:

**Example 13** *Two directly nested do loops:*

```

!HPF$ INDEPENDENT
  DO i=1,n
!HPF$ INDEPENDENT
  DO j=1, m
    A(j,i) = (j-1)*n+i
  END DO
!HPF$ INDEPENDENT
  DO k=m.1.-1
    B(k,i) = A(m-k+1,i)
  END DO
END DO

```

The above loop nest will not be parallelized since two independent loops are present at the same level. To resolve this problem, we can either delete the outer INDEPENDENT directive or one of the inner INDEPENDENT directives. To decide which to choose, we need to take into consideration the communication cost and

degree of parallelism of each solution, as well as how the arrays involved in the loops are distributed. For example, in this project, since the arrays are distributed along the inner loop index, we choose to delete the INDEPENDENT directive for the outer do loop. By doing this, the inner loop index can be distributed in the same way as the computations contained in it. Thus, the whole loop can be distributed among processors and can be executed in parallel. Details of the data distribution will be described in the following section.

### 6.3 Data Mapping

In HPF, computations are partitioned by applying the owner-computes rule. This rule causes the computation to be partitioned according to the distribution of the assigned portion of the computation, which involves localization based on the left-hand-side of an array assignment statement. Therefore, the data distribution over processors determines how computations are partitioned. After computation is partitioned, non-local values are communicated, as necessary, for each computation. Non-distributed values are replicated by the compiler across all processors.

The data mapping strategies used in this project include handling both distributed arrays and compiler replicated arrays (i.e. nondistributed data).

#### 6.3.1 Distributed Arrays

In this project, we used DISTRIBUTE and ALIGN directives in data mapping.

After independent loops are recognized in the data parallelism step, data mapping

is focused on the arrays involved in these independent loops. First, a *home array* needed to be found for each INDEPENDENT loop. A home array is used by the PGHPF compiler to localize loop iterations for an INDEPENDENT loop nest. The indices of the INDEPENDENT loop are associated with dimensions of the home array. Thus, a home array should reference valid array locations for all values of the INDEPENDENT indices. A home array can either be declared by the programmer using the ON HOME clause in INDEPENDENT directives (as shown in the following example), or, if it is not specified that way, the compiler will select a suitable home array from array references within the INDEPENDENT loop.

**Example 14** *ON HOME clause:*

```

    DIMENSION A(n, m)

    !HPF$ DISTRIBUTE A(BLOCK,*)

    !HPF$ INDEPENDENT, ON HOME(A(i,:))

    DO i=1, n

        A(i,:) = i

    END DO

```

After home arrays are found, they are usually distributed along the INDEPENDENT loop indices. Then the other arrays in the loop structure are aligned to the home array according to the computations.

Intuitively, we would think that as more dimensions of the array are distributed, we would attain a higher degree of parallelism. However, when distributing home



arrays, this is not always true. Sometimes, distributing a home array on more dimensions will mean more communication cost in replicating the arrays that are aligned to the home array. The timing result of this project also verifies that for some arrays, when fewer dimensions are distributed, the timing of the program improves. Additionally, for programs containing highly distributed arrays, the number of processors must be chosen carefully, otherwise, the compiler will often get confused on how to handle the distribution and may dump core during run time.

When we were distributing data in the space transform subroutines, we noticed another problem – one array is often involved in different INDEPENDENT loop structures, and in each loop structure, different distributions of the array are required to get the best parallel performance for that INDEPENDENT loop. For the optimal performance in both loops, we would want to distribute the array one way in one INDEPENDENT loop and then redistribute the array another way in another INDEPENDENT loop. However, we found that this often causes dramatic time increase in the program. This is due to the great communication cost caused by the remapping process. Most of the time, a better way to resolve this problem is to sacrifice the performance in the less important INDEPENDENT loops in order to get better parallel performance in the more computationally intensive loops.

### 6.3.2 Nondistributed Data

In both of the space transform subroutines, only about half of the arrays in the subroutines are explicitly distributed or aligned using compiler directives. For

the other half of the arrays whose data mapping patterns are not specified by the programmer, the compiler will by default replicate them across all processors. One reason for not distributing or aligning an array is that there is no obvious relationship between the array and any of the home arrays. But, more often, it is because the array is related to more than one home array and the communication cost of aligning it to any of the home arrays will be greater than the cost of simply replicating it across all of the processors. This is more obvious for small arrays.

Another kind of compiler replicated data are the temporary variables in INDEPENDENT loops. When we use Bernstein's conditions to check the independence of a loop structure, many conceptually independent loops would need substantial rewriting to meet the rather strict requirements for INDEPENDENT. This is caused by the temporary data in the loop which is written and read in more than one iteration. An example of such temporary data is the inner loop index of the nested INDEPENDENT loops. Following is an example of an independent loop that doesn't fit into Bernstein's conditions.

**Example 15** *Do loops containing temporaries S and J:*

```
DO I=1, N
    S = SQRT(A(I)**2 + B(I)**2)
    DO J=1.M
        C(I,J) = S*J
    END DO
```

*END DO*

For this kind of situations, HPF provides the *NEW* clause in the *INDEPENDENT* directive to exclude the compiler replicated loop temporaries from the Bernstein's conditions. When a variable is represented in the *NEW* clause, the loop is treated as if a new instance of the variable is created for each iteration of the *INDEPENDENT* loop, and Bernstein's conditions are discharged. We can still declare the do loop in the above example to be *INDEPENDENT* using the *NEW* clause:

**Example 16** *Using NEW clause to loosen the INDEPENDENT requirement:*

```
!HPF$ INDEPENDENT, NEW(S, J)
```

```
DO I=1, N
```

```
  S = SQRT(A(I)**2 + B(I)**2)
```

```
  DO J=1, M
```

```
    C(I, J) = S * J
```

```
  END DO
```

```
END DO
```

Without the *NEW* clause, one iteration of the above loop may use the values calculated in another loop iteration, which will cause unpredictable results for the program. The *NEW* clause avoids such errors by providing distinct storage units for the temporaries in each iteration of the loop. Thus, the loop can be executed correctly in parallel.

## CHAPTER VII TESTING RESULT AND ANALYSIS

The parallel code was tested for two input files: 2D tokamak and 3D QOS stellarator. The tokamak requires calculation of the magnetic field for 558 Fourier modes. The stellarator input file requires calculation for 11,016 Fourier modes.

### 7.1 2D Tokamak Equilibrium

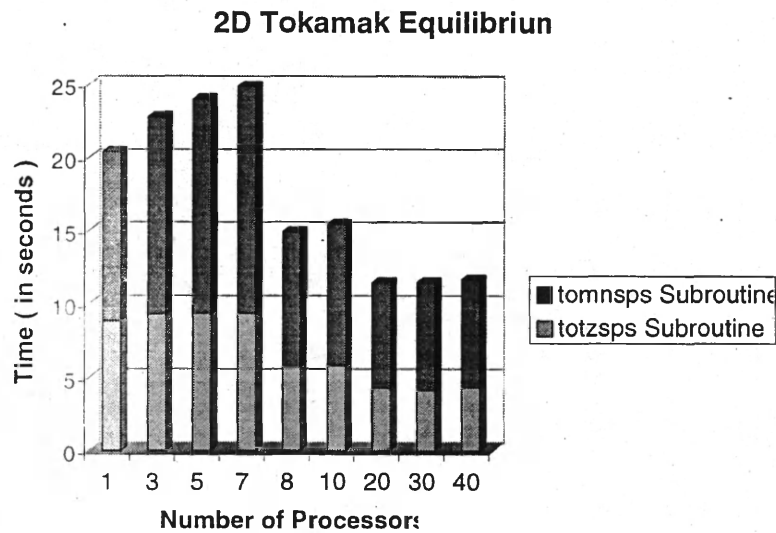


Figure 7.1:

Figure 7.1 shows the timing result for the space transform subroutines, and Figure 7.2 shows the timing result for the whole VMEC program using a 2D tokamak input file.

From the timings in Figure 7.1, we can see that for both of the subroutines, the execution time increases for the first few processors. Then as the number of

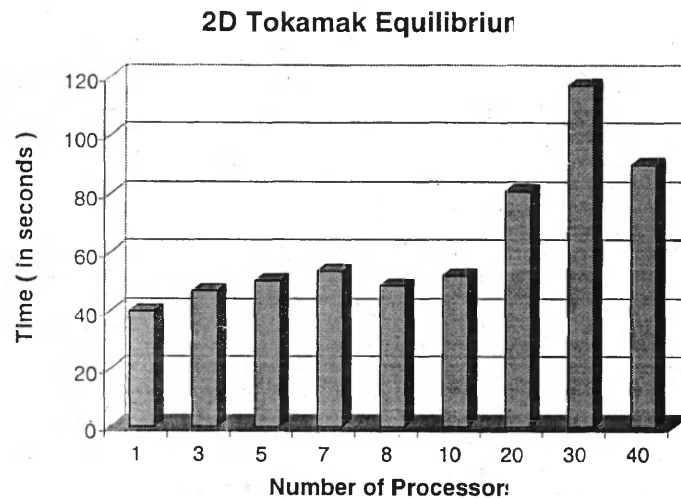


Figure 7.2:

processors continues to increase above eight, the execution time begins to decrease. For the first part of the graph, the reason for the rise in the execution time may be the communication cost caused by the data mapping of HPF. In HPF, no matter how carefully the data is mapped, communication cost caused by the data mapping is almost always unavoidable. For a large problem, the communication cost may be insignificant because of the relatively large speed-up gained by distributing the computation. However, when the problem is small or when the problem is run on a small number of processors, the communication cost caused by data mapping may be significant. And, sometimes, when the communication cost is even greater than the speed-up gained by data mapping, execution time will increase instead of decrease. That is why we saw the first portion of the graph in Figure 7.1 go up.

Figure 7.2 indicates that the execution time increases for the whole program in

spite of the fact that the timing improved in the space transform subroutines. This might be caused by the use of reshape functions, which we mentioned in section 5.3. These reshape functions are used to avoid ineffective distributions of sequence associated arrays. However, by using some of the performance analysis tools on Cray T3E, such as *apprentice*, we can see that such function calls are very time consuming. Especially for small problems like this, it sometimes will take more than half of the execution time of the program. And, when more processors are used, the portion of time spent on the reshape functions can become even higher. Figure 7.3 is an example of a timing result with *apprentice*. For our program, the reshape function calls are made outside the space transform subroutines. Thus, the timing result in Figure 7.1 is not influenced by it. However, for each of the space transform subroutines, there are about 20 reshape functions used in the calling subroutine. These function calls may have caused the program to slow down as shown in Figure 7.2.

One interesting thing illustrated in Figure 7.2 is that there is a peak area around 30 processors. From the more detailed testing result, we found that when the program was run over 29 processors, the compiler threw floating exceptions and the core was dumped. Processor numbers other than 29 worked, but for processor numbers close to 29, the execution time increased dramatically. As the number of processors increases further above 29, the execution time decreases. We have found the same problem for some other input files on certain other processor numbers. We still do not know what caused this phenomenon. But, it may be related to the size and shape of the data distributed in the program and how the compiler handles the distribution.

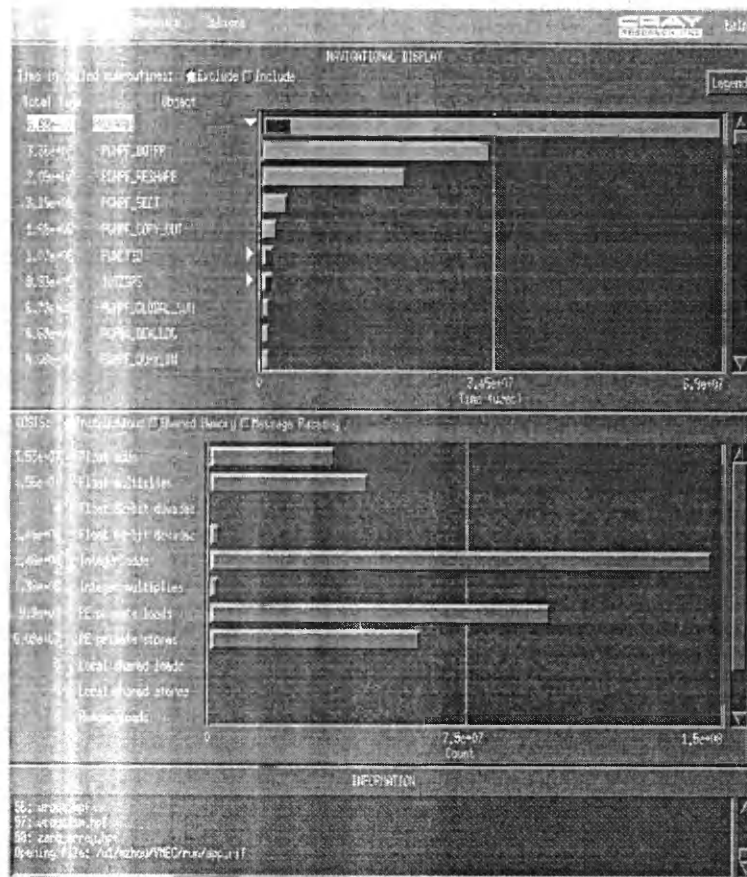


Figure 7.3:

## 7.2 3D QOS Stellarator Equilibrium

Figure 7.4 shows the timing result for the space transform subroutines and Figure 7.5 shows the timing result for the whole VMEC program. The input file used in this test is much larger than in the previous test, and the result is a little different, too. For this input file, the maximum number of processors we can use is 8 due to the CPU time limit on the Cray T3E.

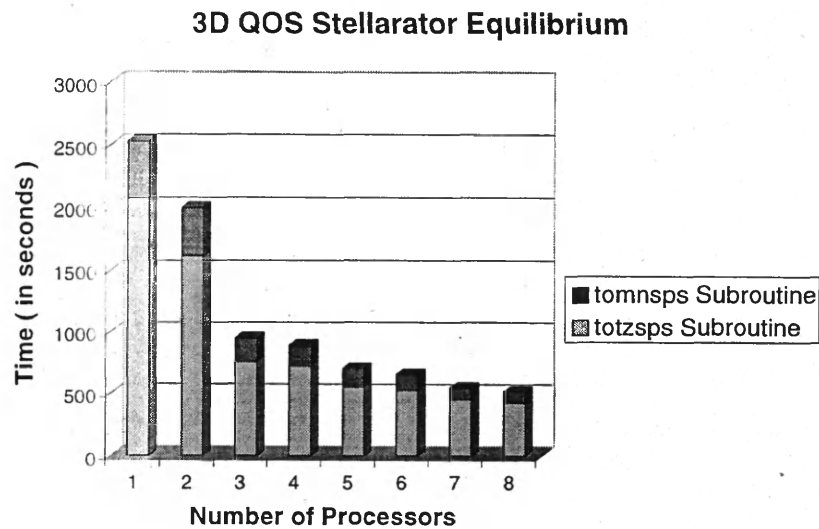


Figure 7.4:

From Figure 7.4, we can see that as the number of processors increases, the timing improves in both of the subroutines. And, when using a small number of processors, the speed-up of the subroutines is more obvious.

Figure 7.5 shows that for a 3D stellarator instead of 2D tokamak, the performance of the program improves as more processors are used.

### 7.3 Conclusion

By comparing Figure 7.5 and Figure 7.2 we can see that the speed-up of the program is greater for 3D stellarators. This is because, for a large problem, the communication cost is insignificant compared to the speed-up gained from distributing the computation. Thus, HPF and, parallel computing, in general is more efficient for large problems. And, for small problems, the relatively large communication cost will



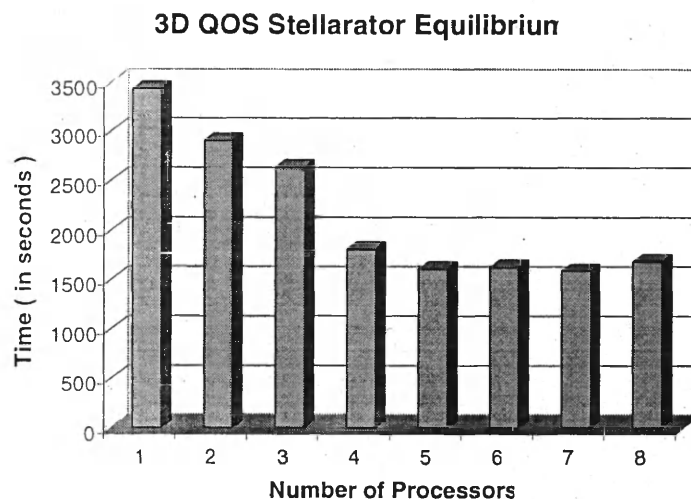


Figure 7.5:

sometimes cause the program to slow down instead of speed up, especially when a small number of processors are used. From our test result, the maximum speed-up is about two for the 3D stellarator input file (as shown in Figure 7.5).

The major restriction that prevents us from further improving the timing of VMEC is the serial bottleneck. According to Amdahl's law, if some fraction  $1/s$  of a program's total execution time executes sequentially, then the maximum possible speedup that can be achieved on a parallel computer is  $s$ . In our program, the parallel part of the code is the two space transform subroutines, which take about 40% of the program's total execution time. And, in these two subroutines, only about 90% of the code is parallelized. Therefore, the maximum speed up of the program can not be more than two no matter how many processors we use. Our test result in Figure 7.5 is consistent with Amdahl's Law. The following diagram shows the code

structure of the parallel VMEC:

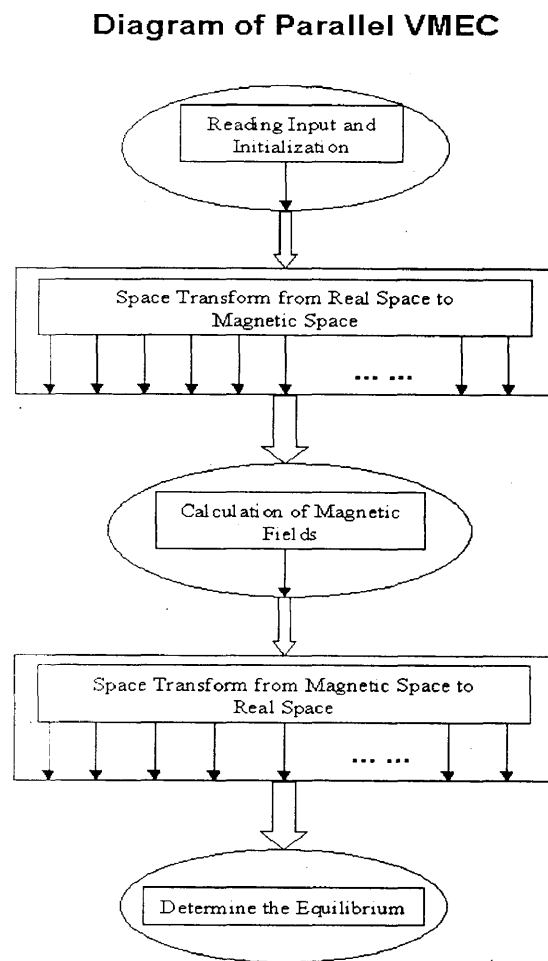


Figure 7.6:

Amdahl's law can also help explain another phenomenon in the above figures. We can see in Figure 7.1, 7.4 and 7.5, the ends of the curves' speed of going down is slowed down compared to the first parts of the curves. This is because according to Amdahl's law, there is a maximum speed-up for each parallel program. And, the performance of the program can not be improved without limit by using more processors. When

the program's performance gets close to its maximum speed-up, a further increase in the number of processors will no longer speed up the program. Instead, it will result in higher communication cost.

## CHAPTER VIII CONCLUSION

In this project, part of VMEC was parallelized using HPF and the program was ported to the Cray T3E. As a result of this project, the program's performance was improved. This improvement can be divided into two stages: vector modification and parallelization. In the vector modification stage, timing is improved by about 35% by using array operations in Fortran 90; In the parallelization stage, timing is further improved by up to 45% by using HPF. Since this was a study in improving performance of a very complex code, the mechanisms we used in this project are not perfect. Future work can be performed to solve the existing problems and further improve the system's performance.

There are two things that can be attempted in the future work. The first thing is to reduce the serial bottleneck in the current parallel code, which means that the serial portion of the code must be reduced. To do this, more subroutines need to be parallelized besides the two space transform subroutines. However, this will cause the increased use of reshape functions, which will add extra execution time to the program. To resolve this problem, it is necessary to find a more efficient way for passing arguments than using reshape functions.

The second thing is to port the VMEC optimizer to parallel structure. The current optimizer calls the equilibrium solver repeatedly with different input parameters and then finds out which one is closest to the target plasma. This process can be parallelized by making different processors run the equilibrium solver with different

input parameters at the same time, as shown in Figure 8.1. One advantage of this approach will be the low communication cost between processors. Since each processor will be running the same program with its own input parameters, their execution is relatively independent of each other. Only the *result* of each processor is collected and compared at the end of the optimizer, and there will be little communication between processors during the execution of the equilibrium solver.

### Parallelization of VMEC Optimizer

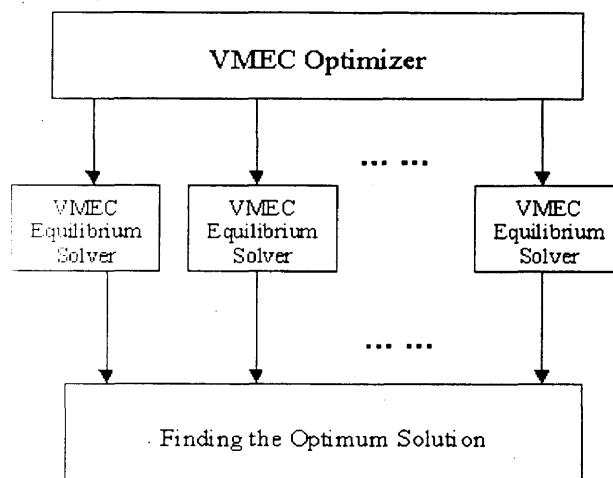


Figure 8.1:

## REFERENCES

- [1] George S. Almasii, Allan Gottlieb, Highly Parallel Computing(Second Edition), The Benjamin/Cummings Publishing Company, Inc., 1994
- [2] Nicholas Carriero, David Gelernter, How to Write Parallel Programs, The MIT Press, Cambridge, 1990
- [3] Richard Dendy, Plasma Physics: An Introductory Course, Cambridge University Press, 1993
- [4] Ian Foster, Designing and Building Parallel Programs, Addison-Wesley Publishing Company
- [5] S. P. Hirshman, *et al.*, Three-Dimensional Free Boundary Calculations Using a Spectral Green's Function Method, Computer Physics Communications 43(1986) 143-155
- [6] Charles H. Koelbel, *et al.*, The High Performance FORTRAN Handbook, The MIT Press Cambridge, Massachusetts, London, England, 1994
- [7] Vipin Kumar, *et al.*, Introduction to parallel computing: design and analysis of parallel algorithms, The Benjamin/Cummings Publishing Company, Inc., California, 1994
- [8] Ewing Lusk, Ross Woverbeek, *et al*, Portable Programs for Parallel Programs, Holt, Rinehart and Winston, Inc, New York, 1987
- [9] Michael Metcalf, John Reid, Fortran 90 Explained, Oxford University Press, United Kingdom
- [10] Larry R. Nyhoff, Sanford C. Leestma, FORTRAN90 For Engineers & Scientists, Prentice Hall, Upper Saddle River, New Jersey
- [11] Gregory F. Pfister, In Search of Clusters: The Coming Battle in Lowly Parallel Computing, Prentice Hall PTR, New Jersey, 1995
- [12] D. A. Spong, S. P. Hirshman, *et al.*, Design Studies of Low-Aspect Ratio Quasi-Omnigenous Stellarators, Phys. Plasmas 5, 1752(1998)
- [13] Gary Sabot, High Performance Computing: Problem Solving with Parallel and Vector Architectures, Addison-Wesley Publishing Company, Inc., 1995
- [14] Gregory V. Wilson, Practical Parallel Programming, The MIT Press, London, 1995