MasPar Fortran User Guide

Software Version 2.0

Document Part Number 9303-0100
Revision A4, July 1992

MasPar Computer Corporation
Sunnyvale, California

MasPar Fortran User Guide
Part Number 9303-0100, Revision A3
Software Version 2.0, 20 July 1992

MasPar Computer Corporation makes no representation or warranties regarding the contents of this document and reserves the right to revise this document or make changes in the specifications of the product described herein at any time without obligation to notify any person of such revision or change.

MasPar Computer Corporation
749 North Mary Avenue
Sunnyvale, California 94086
phone: 408-736-3300
FAX: 408-736-9560

Preface

MasPar Fortran is MasPar Computer Corporation's version of the well-known Fortran 77 programming language with array processing features added from the Fortran 90 ISO standard, and other enhancements from DEC Fortran. These additions give the programmer effective use of the data-parallel processing capabilities of the MasPar® system.

Writing successful programs in MasPar Fortran requires not only an understanding of the syntax of the MasPar Fortran language; it requires an understanding of parallel programming concepts and the manner in which the MPF compiler takes advantage of the MasPar machine. This user guide introduces array mapping and communication, describes the unique array extensions of MasPar Fortran, provides guidelines for developing MasPar Fortran programs, and describes how to compile and run your programs. The complete details of the MasPar Fortran language are described in the *MasPar Fortran Reference Manual.*

MasPar Computer Corporation also provides another language, MPL (MasPar Programming Language), with which you can write programs for the MasPar data-parallel processor. MPL, MasPar's lowest level programming language, gives the programmer direct control over the machine. Chapter 7 describes how to call MPL routines from within a MasPar Fortran program. See the *MasPar Programming Language (MPL) Reference Manual* for details on that language.

NOTE: Be sure to read the *MasPar Fortran Release Notes* for details concerning any special restrictions in this release.

Who Should Use this Book

This user guide is intended for programmers and engineers who have a basic understanding of the Fortran language. This book assumes you are familiar with programming in Fortran 77 and have some knowledge of parallel programming concepts. Chapter 7 of this book assumes you are proficient with the MPL programming language.

Because the front end of the MasPar system runs the ULTRIX operating system (the DEC version of the UNIX operating system), basic knowledge of that is also useful and assumed.

How this Book Is Organized

The *MasPar Fortran User Guide* is organized into the following parts:

- Chapter 1 introduces MasPar Fortran and identifies some of the features that set it apart from Fortran 77.
- Chapter 2 describes data allocation and array mapping in MasPar Fortran.
- Chapter 3 describes communication in MasPar Fortran: between the front end and the DPU, and among PEs.
- Chapter 4 provides tips and suggestions for developing MasPar Fortran programs and improving performance.
- Chapter 5 describes how to go about converting Fortran 77 programs to MasPar Fortran if you do not have MasPar VAST-2.
- Chapter 6 describes how to compile your MasPar Fortran programs.
- Chapter 7 discusses how to call other languages from a MasPar Fortran program.
- Appendix A is a glossary of MasPar Fortran terms.
- Appendix B provides example programs.
- Appendix C lists I / O error messages.

Conventions Used in this Book

UPPERCASE	Fortran keywords appear in UPPERCASE letters.
`typewriter`	Examples of MasPar Fortran code appear in `typewriter font`. Enter text exactly as shown.
italics	Parameters provided by users are shown in *italics*.

[ ]	Optional elements are enclosed in square brackets.
...	Ellipses indicate that an item can be repeated one or more times.
boldface	**Boldface font** is used to highlight new terms and concepts when they are first defined.

Related Publications

MasPar Fortran is designed to run on the MasPar family of computers with the DEC ULTRIX operating system as a front end. For complete documentation of Fortran for ULTRIX and the runtime libraries, see the documentation provided by Digital Equipment Corporation.

You should also have the following MasPar manuals:

- *MasPar Fortran Reference Manual*, PN 9303-0000
- *MasPar System Overview*, PN 9300-0100
- *MasPar Programming Environment (MPPE) User Guide*, PN 9305-0000
- *MasPar Programming Language (MPL) User Guide*, PN 9302-0101
- *MasPar Programming Language (MPL) Reference Manual*, PN 9302-0001
- *MasPar Commands Reference Manual*, PN 9300-0300
- *MasPar Architecture Specification*, PN 9300-5001

Also, be sure to read the *MasPar Fortran Release Notes* for the current release before using the MasPar Fortran compiler.

The array processing extensions that are implemented in this version of MasPar Fortran are based on the Fortran 90 ISO standard. Future enhancements to MasPar Fortran may implement other aspects of this standard. For more information about Fortran 90 (previously referred to as Fortran 8x), MasPar Computer Corporation recommends the following books:

- *Fortran 90 Explained*, 1st edition, by Michael Metcalf and John Reid, Oxford Science Publications, Oxford University Press, 1990.
- *Programmer's Guide to Fortran 90*, by Brainerd, Goldberg, and Adams, Intertext Publications, McGraw-Hill Book Co., 1990.

Getting Help

To get online help on the MPF compiler, type

```
man mpfortran
```

at the system prompt.

To get a list of all the available MasPar man pages, type

```
man -k maspar
```

or

```
apropos maspar
```

at the system prompt.

To get more general help on how to program the MasPar parallel processing system, study the examples in $MP_PATH/examples.

Contacting MasPar

To report defects or make comments on MasPar products and documentation, you can contact MasPar via email, phone, or FAX at these locations:

In North America and the Pacific Rim:

Customer Support
MasPar Computer Corporation
749 North Mary Avenue
Sunnyvale
California
94086
USA

phone: 1-800-526-0916
hours: 8AM - 6PM PST

FAX: 408-736-9560
email: support@maspar.com

In Europe and the United Kingdom:

European Customer Support
MasPar Computer Ltd
8 Commerce Park
Brunel Road
Theale
Reading RG7 4AB
Berkshire, England

phone: +44-734-305444
hours: 9AM - 5PM BST

FAX: +44-734-305505
email: support@mpread.co.uk

Table of Contents

Preface

Who Should Use this Book ii
How this Book Is Organized ii
Conventions Used in this Book ii
Related Publications iii
Getting Help iv
Contacting MasPar iv

Chapter 1. Introduction to MasPar Fortran 1-1

MasPar System Overview 1-2
- MasPar System Terminology 1-2
- Benefits of MasPar Fortran 1-3

Features of MasPar Fortran 1-4
- Array Constructs 1-4
  - Whole Arrays 1-5
  - Array Sectioning 1-6
  - Array Constructors 1-8
- Array Assignments 1-9
  - Assignment Statements 1-9
  - WHERE Statement and Construct 1-9
  - FORALL Statement 1-10
- Array Expressions 1-10
- Automatic Arrays 1-11
- Intrinsic Functions 1-11
- Attribute Specification 1-12
- Control Structures 1-12

Chapter 2. Data Allocation and Array Mapping 2-1

Data Allocation 2-1
- Controlling COMMON Array Data Allocation 2-2
- Passing Arrays Between the Front End and the DPU 2-3
- ONDPU and ONFE Compiler Directives 2-4

Default MasPar Fortran Array Mapping 2-6
- Mapping MasPar Fortran Arrays 2-6
- The PE Grid 2-7
- Mapping Directives 2-18
- Array Alignment 2-24
- Performance Ramifications 2-25

Chapter 3. Communication 3-1

Front End-to-DPU Communication 3-2

PE-PE Communication 3-4

Chapter 4. Developing MasPar Fortran Programs 4-1

Programming Tips 4-2
- Arrays and Indexed Array Elements 4-2
- Aliasing 4-3
- Argument Copying on a Subroutine Call 4-5
- Storage Management 4-6

Compile-Time Analysis 4-7
- Array Sloshing between Front End and DPU Arguments 4-9
- Global Router Operations 4-10
- X-Net Operations 4-10
- Scalar Sloshing between the Front End and the DPU 4-11
- DPU Array Element Access 4-11
- Scalar FORALL Processing 4-12
- Scalar Array I / O 4-13
- Strip Mining 4-13
- Loop Fusion 4-13
- Blocked Algorithm Inhibition 4-14
- Storage 4-14

Execution Profiling 4-15

Runtime Error Checking of Array Mapping 4-16

Timing Program Performance 4-17

Table Lookup with FORALL 4-18

Getting High Performance I / O 4-19

Chapter 5. Converting Fortran 77 Programs 5-1

Using Conversion Tools 5-2

Converting Your Program Manually 5-2
Preparing to Run Your Program on the Front End 5-2
Preparing Your Program to Handle COMMON Arrays 5-3
Converting Your Program to Use Fortran 90 Array Features 5-4

Chapter 6. Compiling MasPar Fortran Programs 6-1

Functions of the Compiler 6-2

The mpfortran Command 6-2
Syntax 6-2
Specifying Input Files 6-3
Specifying Options on the Command Line 6-3

Specifying the Target PE Array Size 6-6

Compiling with Optimization 6-7

Floating Point Faults 6-7

Chapter 7. Calling Other Languages 7-1

Calling Front-end Routines from MasPar Fortran 7-2
User-Written Front-end Routines 7-2
Identifying Front-end Routines to the MPF Compiler 7-2
Some Example Calls 7-3
Input / Output 7-5
Front-end C Considerations 7-5
Character Variables with C and Fortran 7-6
Front-End Fortran Libraries 7-7

Calling MPL from MasPar Fortran 7-8
Identifying MPL Routines to the MPF Compiler 7-8
Argument Blocks 7-9
Passing Data Between MasPar Fortran and MPL 7-11
Mapping Arrays onto the PE Grid 7-12
Mapping MasPar Fortran Arrays 7-12
The Active Set 7-13
Allocating Arrays that Are Larger than the Machine 7-15
Accessing Front-End Scalar Arguments from MPL 7-26
Accessing MasPar Fortran Character Variables 7-30

Appendix A. MasPar Fortran Terms A-1

Appendix B. Example Array Operations B-1

Arithmetic Operations B-2

Conditional Expressions in MasPar Fortran B-7

Array Movement Operations B-10
EOSHIFT and CSHIFT Intrinsic Functions B-10
SPREAD Intrinsic B-13

Array Reduction Operations in MasPar Fortran B-16
Vector and Matrix Operations B-19

Appendix C. I / O Error Messages in MasPar Fortran C-1

Index

List of Figures

Figure 2-1 Virtual Layers into PE Memory 2-7
Figure 2-2 [x,y] PE Coordinates 2-8
Figure 2-3 One-Dimensional Cut and Stack 2-9
Figure 2-4 Cut and Stack for Two-Dimensional Arrays 2-10
Figure 2-5 Mapping Array [225,9] on a 4K Machine 2-11
Figure 2-6 Example 1: a = 0.0 2-12
Figure 2-7 Example 2: a(:, :, 1) = 0.0 2-13
Figure 2-8 Example 3: a(1, :, :) = 0.0 2-14
Figure 2-9 Example 4: a(:, 1:8, :) = 0.0 2-15
Figure 2-10 Example 5: a(1, 1, :) = 0.0 2-16
Figure 2-11 Example 6: a(8, :, 1:4) = 0.0 2-17
Figure 2-12 Mapping Example 1: Canonical / XBITS, YBITS 2-20
Figure 2-13 Mapping Example 2: ALLBITS, MEMORY 2-21
Figure 2-14 Mapping Example 3: PACKBITS:1, PACKBITS:2 2-22
Figure 2-15 Mapping Example 4: 0:4, 5:11 2-23
Figure 3-1 Two-Dimensional Array Misalignment 3-4
Figure 7-1 Front End and ACU Argument Blocks 7-10
Figure 7-2 Mapping Two-Dimensional Arrays Using Cut and Stack 7-16
Figure 7-3 Layers of a "Virtual" Machine 7-18
Figure 7-4 Mapping a 40x40 Array 7-24
Figure 7-5 Copying Front-End Values to the ACU 7-27

Chapter 1

Introduction to MasPar Fortran

MasPar Fortran is based on Fortran 77 with extensions from DEC Fortran and the Fortran 90 ISO standard. The most important of these extensions are the Fortran 90 array statements, with which you can take advantage of the data-parallel features of the MasPar machine. These extensions include array notation and expressions, arrays as first-class objects, array sectioning, vector-valued subscripts, array constructors, control structures (CASE, DO, DO WHILE), a WHERE construct, and double-complex data types. Additionally, many new intrinsic functions have been added.

MasPar Fortran supports Fortran 77 as a subset. Fortran 77 programs that meet certain restrictions can be run "as is" on the MasPar system. However, such programs will not make use of the parallel-processing capabilities of the DPU. Although some DPU housekeeping code is generated, when you compile a Fortran 77 program most of the code will be scalar front-end code. As a result, when the program is executed it will execute at the speed of the front end. To use the DPU effectively, you need to modify certain aspects of your program. By using Fortran 90 array statements (which are compiled into DPU code) you can take advantage of the power afforded by the data-parallel hardware.

To program effectively in MasPar Fortran you should

- have an understanding of the basic features of the MasPar architecture,
- understand how MasPar Fortran manages the interface between the front-end and the data-parallel unit (DPU), and

- understand how the PE grid is used for large-scale scientific computation.

Although very detailed knowledge of these concepts is not necessary, some understanding of them will help you program more effectively in MasPar Fortran. In addition to the MasPar Fortran programming manuals, you should review the *MasPar Architecture Specification* and the *MasPar Programming Language (MPL) Manuals*.

This chapter provides a brief introduction to the MasPar system and the MasPar Fortran high-level programming language.

MasPar System Overview

The complete MasPar system includes a massively parallel, data-parallel processor (the data parallel unit or DPU), a front-end graphics workstation that runs a version of the UNIX operating system, programming languages customized to use the DPU effectively, and tools for debugging and analyzing how well your program uses the DPU.

MasPar Fortran is one of two languages currently available with the MasPar system. The other is MPL (MasPar Programming Language), a language developed by MasPar specifically to program the DPU. For more information on MPL, see the *MasPar Programming Language (MPL) Reference Manual* and *User Guide*.

MasPar System Terminology

You do not need a detailed understanding of the MasPar system architecture to program in MasPar Fortran. However, it is important that you understand some of the basic concepts of the MasPar system and parallel programming before attempting to write MasPar Fortran programs. Following are some basic terms relative to the MasPar system. For more information on the MasPar system, see the *MasPar System Overview* and the *MasPar Architecture Specification*.

active set	The set of PEs that is enabled at any given time during program execution.
ACU	Array Control Unit; part of the DPU, the ACU is a processor with its own data and instruction memory. It controls the PE array and performs operations on singular data.
data parallel	Descriptive of the manner in which the MasPar system (and other SIMD machines) processes instructions. Program instructions are executed serially, but instructions that operate on parallel data are executed by each parallel data processor simultaneously.
DPU	Data Parallel Unit; that part of the MasPar system where all the parallel processing is performed. The DPU includes the ACU, the PE array, and PE communications mechanisms.
front end	That part of the MasPar system that runs an implementation of the UNIX operating system on a graphics workstation with windowing capabilities.

massively parallel	Descriptive of any computing system that has at least 1000 processors acting in parallel, such as the MasPar system.
MPPE	The MasPar Programming Environment on the MasPar system. It is a set of integrated, graphical tools that help you run, analyze, and debug your programs. For more information on the MPPE and the tools provided with it, see the *MasPar Programming Environment (MPPE) User Guide.*
PE	Processor element. Each PE is a load/store arithmetic processing element with dedicated register space and RAM. Each PE receives the same instruction simultaneously from the ACU. PEs that are enabled then execute the instruction on variables that reside on the PEs. The set of PEs that is enabled at any given time during program execution is called the active set.
PE array	A two-dimensional matrix of at least 1024 processor elements (PEs).
SIMD	Single instruction, multiple data. See *data parallel.*

Benefits of MasPar Fortran

Two significant features of the MasPar Fortran compiler are

- the management of an asynchronous front end/DPU interface, and
- the virtualization of arrays larger than the physical size of the PE grid.

Both features are automatically implemented by the MPF compiler; that is, they require no special directives or coding techniques. Also, you do not need to be concerned that your array sizes might exceed the size of the PE grid. Any virtualization required is automatically implemented by the compiler.

Another feature of the MasPar Fortran compiler is that it takes advantage of

- the MasPar load/store architecture with its large number of registers.

Because register operations can often happen at much greater speed than memory-to-memory operations, the result is substantial performance gains. However, to realize these gains, you must structure operations to use the register space to its full potential. The optimized MPF compiler technology minimizes register "spills" to memory. This is different from MPL, which requires that you decide and explicitly declare which variables are the best candidates to maintain in registers.

Although the MPF compiler handles these things, it takes much of its direction from how you write your code. This book provides the background necessary to help you write effective MasPar Fortran programs.

Features of MasPar Fortran

MasPar Fortran provides many array extensions to standard Fortran 77. In Fortran 77, you program operations on arrays element by element, using iterative DO loops. In MasPar Fortran, you can write array calculations and matrix operations as simple expressions rather than iterative loops. The MPF compiler allocates these calculations on the DPU and generates data-parallel code for them.

With MasPar Fortran, you can use whole arrays and array sections (as well as scalars) as operands. An expression can evaluate to a scalar value or an array value.

In addition to the assignment statement available with Fortran 77, MasPar Fortran includes conditional (masked) array assignments, using a WHERE statement or WHERE construct, and element array assignments, using the FORALL statement.

MasPar Fortran allows arrays and array sections as arguments and as results of functions.

MasPar Fortran provides numerous new intrinsic functions: array inquiry functions, which return information about an argument based on its declaration; transformational functions, which have arrays as arguments, as results, or both; and elemental functions, which can have both scalar and array arguments.

In addition, MasPar Fortran provides a CASE construct, extensions to the DO construct, and a DO WHILE construct.

Array Constructs

Fortran 77 defines these concepts:

- **Scalar element:** A single element; for example, one integer or one real number.
- **Array:** A series of scalar elements in one or more dimensions. A single element of an array is equivalent to a scalar element and is referenced by subscripting the array name. The number of dimensions is called the **rank** of the array.

There are three types of array objects available with MasPar Fortran:

- whole arrays
- array sections
- array constructors

If two array objects have the same shape they are considered **conformable**; that is, they have the same rank, and each dimension contains the same number of elements. Array objects must be conformable to be operated on together. A scalar value is conformable to any array; during an operation, the scalar is treated as an array in which each element has the scalar value.

Whole Arrays

To manipulate arrays in Fortran 77, you must use a series of scalar operations, such as an iterative DO loop, except when performing I/O and passing arguments. For example, a matrix addition in Fortran 77 would be coded as follows:

```
      REAL A(10, 10), B(10, 10), C(10, 10)
      . . .
      DO 30 I = 1, 10
          DO 30 J = 1, 10
              A(I, J) = B(I, J) + C(I, J)
30    CONTINUE
```

When compiled on the MasPar system, this code fragment would execute serially on the front end.

Because the Fortran 90 array extensions allow you to perform operations on whole arrays as if they were single objects, the above operation can be recoded in MasPar Fortran much more simply, as follows:

```
REAL A(10, 10), B(10, 10), C(10, 10)
   . . .
   A = B + C
```

The MPF compiler allocates this recoded segment on the DPU, where it executes in parallel fashion.

Assumed-Shape Arrays

An **assumed-shape array** is a dummy argument array; it assumes the shape of the associated actual argument array. Each dimension is noted by a colon; the upper bound is never specified, and the lower bound can be specified, but is not required. The extent of each dimension of an assumed-shape array is the extent of the corresponding dimension of the actual argument array. This example declares assumed-shape arrays:

```
SUBROUTINE TEST (A, B, C, M)
    REAL A(:), B(0:), C(M:,:)
```

The value of the upper bound of a dimension of an assumed-shape array is the value of the extent of the associated actual argument plus the lower bound, minus one. (See the UBOUND intrinsic function, described in Chapter 9 of the *MasPar Fortran Reference Manual*.)

You can declare assumed-shape arrays with a specification expression in the lower bound of any dimension.

Array Sectioning

An array section is a subset of the elements of an array and is itself an array (a **subarray**). You refer to the section by means of a section specifier:

- subscript triplet notation
- vector-valued subscripts

Subscript Triplet Notation

You can refer to an array section by specifying a range of subscript values with a **lower bound**, an **upper bound**, and the increment, or **stride**, between each value. These values are separated by colons. The stride can be a positive or negative value but cannot be zero. This notation is called a **subscript triplet**; it is a concise notation for an iterative DO loop. The following program example shows the use of array sectioning, using the subscript triplet notation with a positive stride.

```
INTEGER SIZE
PARAMETER (SIZE = 20)
REAL X(SIZE/2), Y(SIZE), Z(SIZE)
X = Y(1:SIZE:2) * Z(2:SIZE:2)
```

The triplet notation Y(1:SIZE:2) means take every second element (2 is the stride) of array Y, starting at the first element (1, the lower bound), and going to the last element (SIZE, the upper bound).

This program takes the odd elements of Y and the even elements of Z, then multiplies the corresponding elements of Y and Z, and stores the result in X. Arrays Y and Z (which in this case are one-dimensional arrays) are twice as long as array X (also a one-dimensional array).

The same kind of selection could be written in a Fortran 77 DO loop:

```
DO 10 I = 1, SIZE, 2
```

However, the resulting calculation would not be executed in parallel, because Fortran 90 syntax is not used.

Suppose the stride used is negative; –2, for instance. The subscript range will start with the value of the first subscript (1) and decrement by the absolute value of the stride (2) until the smallest value greater than or equal to the second subscript (SIZE) is reached. However, in this case the range would be empty, since the value of the second element (SIZE) is greater than the value of the first element (1).

You can easily specify all elements along a dimension with the triplet notation. The following example shows a row of array A multiplied by a column of array B. The result is summed to a single value (with the MasPar Fortran intrinsic function SUM):

```
C(J, I) = SUM( A(:,J) * B(I,:) )
```

When you specify array sections, you can omit some of the values of the subscript triplet.

Omitting the first or second element causes the compiler to assume the declared value of the lower or upper bound, respectively. Omitting the third element causes the stride to default to a value of 1.

Vector-Valued Subscripts

A **vector-valued subscript** is a vector, or a rank one array-valued expression, used to index a dimension of an array. A vector-valued subscript can define a many-to-one section, where one or more elements of the vector expression have the same value. Vector-valued subscripts can be used on either the right or left of an assignment. However, a many-to-one subscript cannot appear on the left.

When an array is subscripted with a vector, the resulting shape has an extent due to the vector-valued subscript, which is the size of the vector. In the example below, the expression is conformable because the vector *v* is the same size as the left side array *r*.

```
integer a(10)
integer v(5), r(5)
data a /21, 22, 23, 24, 25, 26, 27, 28, 29, 30/
data v /2, 4, 10, 8, 10/

  r = a(v)
end
```

If this code is compiled and executed, *r* will contain [22, 24, 30, 28, 30].

Vector-valued subscripts can also be used together or with other sections to index higher rank arrays. For example:

```
integer, dimension( 5 ) :: v
integer, dimension( 5, 5 ) :: a, rslt
data v /5, 4, 3, 2, 1/

call init( a )
  rslt = a(v, 5:1:-1)
end
```

The expression *a(v, 5:1:-1)* is a rank two array with the shape [5,5]. In this example *v* is initialized to [5, 4, 3, 2, 1]. If the array *a* is initialized to:

```
 1   2   3   4   5
 6   7   8   9  10
11  12  13  14  15
16  17  18  19  20
21  22  23  24  25
```

then *rslt* will contain:

```
25  24  23  22  21
20  19  18  17  16
15  14  13  12  11
10   9   8   7   6
 5   4   3   2   1
```

In Fortran 77, the preceding vector-valued subscript assignment would be written as follows:

```
do j = 1, 5
   do i = 1, 5
     rslt(i, j) = a(v(i), 6-j)
   end do
end do
```

In contrast to the Fortran 77 code, the vector-valued subscript operation takes place in parallel. Vector-valued subscript operations use the global router of the DPU.

Array Constructors

Using an **array constructor**, you can form an array value in an executable or data initialization statement by listing the elements. This list is a sequence of scalar values, interpreted as a one-dimensional array. The array element values are those specified in the sequence, as if an assignment were made for each element. You can specify this sequence of values as either individual scalar values, ranges of values, or a one-dimensional array.

You delimit the array constructor by parentheses with a slash just inside each parenthesis.

The data type of an array constructor is the data type of the first item in the list; all elements of the constructor must have the same data type as the first element.

In this example, the individual scalar values of the elements of A4 are specified:

```
REAL A4 (4)
    A4 = (/ 32.1, 83.5, 90.6, 22.1 /)
```

If the sequence were empty, a zero-sized array would be constructed.

Here an implied do specifies a range of values and a stride:

```
INTEGER I4 (6)
    I4 = (/ (I, I=32, 42, 2) /)
```

You can create array objects with more than one dimension by using the intrinsic function RESHAPE. For example

```
Y = RESHAPE (SOURCE = (/ I, I=1,6 /), SHAPE = (/ 3,2 /))
```

creates the following values for array Y:

```
Y(1,1) = 1
Y(2,1) = 2
Y(3,1) = 3
Y(1,2) = 4
Y(2,2) = 5
Y(3,2) = 6
```

Array Assignments

Variables become defined or redefined by **assignments.** MasPar Fortran allows you to make assignments in the following ways:

- assignment statements
- conditional (or masked) array assignments (WHERE statement and WHERE construct)
- element array assignments (FORALL statement)

Assignment Statements

An array assignment statement is one in which the left side of the statement is a whole array or an array section. The expression on the right side of the statement must evaluate to a conformable value; that is, an array of the same shape.

Array assignments are allowed for numeric and logical assignments, but not for character assignments.

As with scalar assignments, before the assignment is made, the value on the right side of the expression is converted to the data type of the variable on the left side, if necessary. If a scalar expression is assigned to an array variable, the scalar value is assigned to each element of the array variable.

When the variable in the assignment is an array, the assignment is made element-by-element; that is, the array element values of the expression are assigned in parallel to the corresponding elements of the array variable.

If the left side of the assignment statement uses a vector-valued subscript, the vector-valued subscript cannot contain duplicate entries.

WHERE Statement and Construct

Sometimes you might want to perform array operations for only certain elements of an array. With the WHERE statement (or WHERE construct), you can make array element assignments conditionally. Depending on the value of a logical array expression appearing in a WHERE statement (or construct), you can **mask** the evaluation of expressions or assignment values in an assignment statement.

The WHERE statement operand is a logical array expression that conforms to the shape of the array variable being defined. This logical expression is evaluated first. Those elements that have the value TRUE then are evaluated. Only the elements of the array variable that correspond to the elements that have the value TRUE in the logical expression are assigned the new value; the other elements of the array variable remain unchanged.

Following are some simple WHERE statements:

```
WHERE (A > 0.0) A = 1.0/A     !  A is an array of type REAL

WHERE (X(:,1:N) < Y(:,1:N)) Y(:,1:N) = X(:,1:N)
```

A WHERE construct begins with a WHERE statement and ends with an END WHERE statement. Every array expression in the block must be conformable with the logical array expression.

An example of a WHERE construct is as follows:

```
WHERE (S .GT. A(1:N))
    A(1:N) = S
ELSEWHERE
    A(1:N) = 0.0
END WHERE
```

You cannot nest WHERE blocks, transfer control into a WHERE block, or end a DO loop in a WHERE block.

FORALL Statement

With the FORALL statement, you specify a parallel array assignment statement in terms of array elements or array sections. Additionally, you can optionally mask the array assignment by adding a scalar logical argument to the FORALL statement.

The FORALL statement is a parallel equivalent to a DO loop. It specifies parallel calculation for the included assignment statement.

In this example, the I,J'th element of X is multiplied by the J,5,I'th element of Y and stored in the I,J'th element of Z:

```
INTEGER X(10,10), Y(10,10,10), Z(10,10)
FORALL (I=1:10, J=1:10)   Z(I,J) = X(I,J) * Y(J,5,I)
```

(The results of this example are not executed in parallel.)

The following examples show the FORALL statement with vector-valued subscripts:

```
FORALL (i=1:N)    A(i) = B(V(i),i)
FORALL (i=1:N, j=1:N)   A(X(i,j),Y(i,j)) = B(i,j)
```

See the *MasPar Fortran Reference Manual* for restrictions on the FORALL statement.

Array Expressions

An **expression** consists of one or more data references called operands that can be combined with predefined operators in computational procedures.

An **array expression** is an expression in which at least one of the operands is a whole array, an array section, or an array constructor, and evaluates to an array with the same shape as the operands.

You can perform any arithmetic, relational, or logical operation on an array expression. The operations are performed on an element-by-element basis between the corresponding elements of the subject arrays.

A **specification expression** appears only in the declaration of arrays. For explicit-shape arrays, the specification expression can be in any bound of any dimension. For assumed-size arrays, the specification expression can be in any bound of any dimension except the upper bound of the rightmost dimension. For assumed-shape arrays, the specification expression can be in the lower bound of any dimension.

Automatic Arrays

A procedure with dummy arguments that are arrays whose sizes (or *extents*) vary from call to call, might require local arrays whose sizes vary. Local arrays whose extents vary in this way are called **automatic arrays.**

An automatic array is dynamically allocated at runtime upon entry to a subprogram, allowing for better use of memory.

An automatic array can appear only in a subprogram. To specify an array in a subprogram as automatic, specify that it is local (not a dummy argument and not in COMMON), and specify its bounds using scalar dummy arguments or variables in COMMON.

The following example shows the automatic array WORK in a subroutine that interchanges two arrays:

```
SUBROUTINE SWAP(A,B)
    REAL, DIMENSION(:) :: A,B
    REAL, DIMENSION(SIZE(A)) :: WORK
    WORK = A
    A = B
    B = WORK
END SUBROUTINE
```

Intrinsic Functions

MasPar Fortran has added many intrinsic functions to the Fortran 77 function libraries, particularly to enhance array operations. These intrinsics fall into the following categories:

- **Inquiry functions** return information about an array argument based on its declaration, not its value. The following example uses the function SIZE to return the number of elements in the array A:

  ```
  J = SIZE(A)
  ```

- **Elemental functions** can have scalar and array arguments. For array processing, at least one of the arguments is an array, and the result is an array; the arguments and the results must be conformable. The function

works element-by-element on the array arguments, computing in parallel the results of the function for each corresponding element of the array. The following example uses the function SQRT to calculate the square roots of N elements of B and stores the results in N elements of A:

```
A(1:N) = SQRT (B(1:N))
```

- **Transformational functions** usually have array arguments, an array result, or both. The result produced has a shape different from the arguments. The following example uses the function SUM to add the values of N elements of array A:

```
SCALAR = SUM (A(1:N))
```

Some intrinsic functions can be referenced with either a specific or a generic name. Only specific scalar names can appear as actual arguments associated with a dummy procedure argument, and when one is called, its arguments must be scalar. If a generic name or a specific name is used to reference an elemental intrinsic function, the shape of the result is the same as the shape of the argument with the greatest rank. If the arguments are all scalar, the result is scalar.

Chapter 9 of the *MasPar Fortran Reference Manual* provides detailed descriptions of all the intrinsic functions available with MasPar Fortran.

Attribute Specification

With MasPar Fortran you specify the attributes of data objects in the type declaration statement of your program using **attributes**. Using the attribute specifier is the same as using the corresponding statement. MasPar Fortran supports five attributes: PARAMETER, INTENT, DIMENSION, OPTIONAL, and SAVE.

See Chapter 3 of the *MasPar Fortran Reference Manual* for a detailed description of these attributes.

Control Structures

MasPar Fortran provides a CASE construct, a DO WHILE construct, and extensions to the Fortran 77 DO construct.

- **CASE Construct**—With the CASE construct you can select and execute a single statement block from a set. See Chapter 5 of the *MasPar Fortran Reference Manual* for a complete description of the CASE construct.
- **DO and DO WHILE Constructs**—With MasPar Fortran you have alternative methods for specifying loop control in a DO construct. Additionally, with the DO WHILE construct, you can repeatedly execute a block of statements while the value of a logical expression is true. See Chapter 5 of the *MasPar Fortran Reference Manual* for a complete description of the DO and DO WHILE constructs.

Chapter 2

Data Allocation and Array Mapping

The MasPar system is composed of a front end and a DPU, each of which has its own processors and memory organization. On the DPU, memory is distributed among each PE in the PE grid. Serial operations are performed on the front end; parallel operations are performed on the PE array in the DPU.

The key to good performance on a data-parallel machine is to use as many PEs as possible and to use them efficiently. This chapter introduces how MasPar Fortran allocates arrays, and how the compiler maps them onto the PE grid. While the MasPar Fortran compiler does this automatically, how you write your code directly influences how the compiler allocates and maps the data. It is important to understand the array mapping so you can choose the most efficient coding techniques for a particular algorithm.

Data Allocation

In MasPar Fortran you do not need to state explicitly which variables should be allocated on the front end and which should be allocated on the PE array. The compiler determines which portions of the program require parallel or serial execution by the way they are used in the program. (Note that the way a variable is *declared* does *not* influence where the compiler places it.) If an array is used in a Fortran 90 manner, it is allocated on the

PE array. You can, however, force a variable to the DPU using the compiler directive, ONDPU. Also, you can keep a non-Fortran 90 variable from unnecessarily moving between the front end and the DPU with the ONFE directive.

It is important to understand how the compiler allocates data so you can minimize data movement between the front end and the DPU. Unnecessary movement slows program performance.

- Local and dummy variables are placed as follows:
  - Numeric and logical arrays that are used in a Fortran 90 manner are placed in the DPU.
  - Other arrays and scalars are placed on the front end.
- With the –fecommon option on (this is the default), all COMMON variables are placed on the front end.
- With the –nofecommon option on, COMMON variables are allocated as follows:
  - Numeric and logical arrays are placed on the DPU.
  - Character arrays and scalars are placed on the front end.
  - If numeric or logical arrays are combined with character arrays or scalars in the same COMMON block, you get an error.

Object files compiled with the –fecommon default and the –nofecommon option cannot be mixed.

Controlling COMMON Array Data Allocation

By default, the compiler allocates all arrays in COMMON block on front-end memory, regardless of how they are used.

The –nofecommon option allocates all COMMON arrays on PE memory, even if they are never used in a Fortran 90 expression. (Character arrays are an exception and are unaffected by the setting of this option. Currently, character arrays are always allocated and processed on the front end.)

Whether you use the –nofecommon option depends on how COMMON arrays are used in your program. If you use COMMON arrays in Fortran 90 array expressions, you *must* use the –nofecommon option or the compiler will flag the array expression with an error. However, if your program uses COMMON arrays in scalar Fortran 77 statements, the compiler will move every element of these arrays from the PE array to the front end, significantly slowing the performance of your program.

NOTE: It is best to avoid any use of COMMON array data in scalar statements and to use COMMON array data only in Fortran 90 statements. Be aware that when you reference COMMON arrays in Fortran 77 code and compile with the –nofecommon option, program performance is seriously degraded.

You can use the ONDPU directive to control placement of named COMMON blocks; see page 2-5.

Passing Arrays Between the Front End and the DPU

Arrays that are used in Fortran 90 expressions (for example, operations on whole arrays or array sections) are mapped onto the PE array. In the example that follows, because A is used in the Fortran 90 statement

```
A = 0
```

it is allocated on the PE array. The fact that A is declared using Fortran 77 syntax does not influence where the array is allocated. Only usage determines its allocation.

```
!
! note that in Fortran 90 "!" comments out
! all characters that follow it
!
subroutine foo( A, M, N)
integer M, N
integer A(M, N)

  A = 0
  do j = 1, N
     do i = 1, M
       a(i, j) = A(i, j) + 1
     end do
  end do
end ! subroutine foo
```

This routine is inefficient because the A=0 forces A to be allocated on the DPU, and its values are then accessed serially in the DO loops. Serial access to DPU-based array elements is even slower than serial access to front-end-based array elements; this should be avoided if possible. The more efficient way to write this routine would replace the DO loop with A=A+1.

In the following example, a one-dimensional, million-element array is declared and later referenced as two-dimensional: A(1000, 1000). This works well in a pure Fortran 77 program. However, problems arise when the Fortran 77 code is mixed with Fortran 90 syntax, because arrays *cannot* be reshaped across Fortran 90 subroutine boundaries. In this example the array A starts out as a one-dimensional, million-element array. Because subroutine foo is a Fortran 77 routine, A can be reshaped across the subroutine call. However, when the Fortran 90 subroutine bar is called, the array declarations on both sides of the call must match.

```
program example
integer A( 1 000 000 )      ! Only Fortran 77 usage
                            ! in the main program
  call foo( A )
end


subroutine foo( A )         ! A Fortran 77 subroutine
integer A( 1000, 1000 )

  call bar( A )
end


subroutine bar( A )         ! A Fortran 90 subroutine
integer A( 1000, 1000 )

  A = A + 1
end
```

The calls in the preceding examples are all legal and execute as expected. It would *not* have been legal to call bar(A) from the main program. The results of mismatched array dimensions and mismatches of array dimension extents across Fortran 90 subroutines are unpredictable. One way to avoid mismatched extents is to use assumed-shape dummy arguments, for example

```
integer, dimension(:)::A
```

In the preceding example, the array A is copied into the DPU at the start of subroutine bar and is *copied back* to the front end at the end of the routine. The copy-back operation is necessary because the contents of A might have changed in the subroutine. This type of copy operation, which involves moving large arrays between the DPU and the front end, severely affects performance.

Each time a routine with front-end actual arguments calls a routine with DPU dummy arguments, these array arguments are copied from the front end into the DPU. At the end of the routine, they are copied back to the front end. To avoid the overhead incurred by these copy operations, revise your program so, whenever possible, array arguments are always on the DPU.

Another way to avoid unnecessary data movement is to force placement of an array on the PE array by performing a Fortran 90 array operation whose only purpose is to place the array on the DPU. Additionally, the ONDPU compiler directive allows you to specify to the compiler that an array argument must be allocated on the DPU.

ONDPU and ONFE Compiler Directives

The ONDPU and ONFE compiler directives allow you to specify whether a given array is allocated on the DPU or on the front end.

The ONDPU compiler directive allows you to specify that a given array is allocated on the DPU (not the front end) even when it is not used in a Fortran 90 expression.

For example:

```
            program TEST
            real, dimension(100,100) :: a,b
CMPF        ONDPU a,b

                 call foo( a,b )
            end
```

In this example, a and b will be allocated on the DPU, even though they are not used in any Fortran 90 array expressions. If the ONDPU directive were not used, they would default to the front end. This example uses local arrays; dummy and automatic array names can also be specified in the ONDPU directive, but COMMON arrays cannot. The ONDPU directive is *not* valid for character arrays.

Be aware, however, that by default the Fortran compiler allocates all COMMON arrays in front-end memory. (This default can be changed with the –nofecommon compiler option; see page 6-4.) The front-end default for COMMON can currently be overridden only by the ONDPU flag for *named* COMMON blocks. This is not true for blank COMMON. For example, if the code below is compiled with the front-end COMMON default, the compiler will issue the error message "Common array illegal in SIMD context with –fecommon".

```
            program ERROR
            integer, parameter :: sizeA = 20, sizeB = 20
            real, dimension( sizeA, sizeB ) :: a, b
            common a, b

CMPF        ONDPU a, b

              call foo( a, b )
            end
```

On the other hand, because the following example uses a named COMMON block, it will compile correctly and allocate the data in */plebeian/* on the DPU.

```
            program TEST2
            real, dimension( 100, 100 ) :: a, b, c, d
            common /foo/ c, d
            common /plebeian/ a, b

CMPF        ONDPU plebeian

            call bar( a, b )
            end
```

Conversely, the ONFE directive allows you to specify that a given array or COMMON block is allocated on the front end, overriding the –nofecommon option. You cannot, however, specify the ONFE directive for an array used in a Fortran 90 construct. You use the ONFE directive to keep the compiler from unnecessarily moving arrays from the front end to the DPU and back again.

Default MasPar Fortran Array Mapping

The default mapping of any array used in Fortran 90 syntax is called **canonical** array allocation. This default MasPar Fortran array-mapping algorithm is efficient for many array operations. You can override this default mapping by using the mapping directives, discussed later on page 2-18.

On the MasPar machine, arrays are mapped onto the PE grid in *columns* and *rows*. The first column of the first row is processor [0,0]. For instance:

4 x 2

y \ x				
	[0,0]	[1,0]	[2,0]	[3,0]
	[0,1]	[1,1]	[2,1]	[3,1]

How processors map on the machine

You can determine the number of PE rows and columns on your machine by executing the mpconfig command from the UNIX command line. Note that this command prints the rows first, followed by the columns, which is the *opposite* of how rows and columns are referred to in MasPar Fortran two-dimensional array declarations.

Mapping MasPar Fortran Arrays

MasPar Fortran arrays are mapped onto the PE grid so that processor [0,0] contains the first Fortran array element. If the array is one-dimensional, it is mapped in a serpentine fashion onto the PE grid. Assuming that the array size is smaller than the machine size, the Fortran element number maps directly onto the iproc number (that is, element 1 -> PE 0, element 2 -> PE 1, ...). (Mapping arrays that are larger than the machine is discussed later.)

In MasPar Fortran, *always* think in terms of *columns* first, then *rows*.

A MasPar Fortran two-dimensional array that will be allocated on the PE grid is declared with the *first* dimension corresponding to the number of *columns* on the machine and the *second* dimension corresponding to the number of *rows*. For instance, in MasPar Fortran the array A(4,2) would be mapped onto the PE grid as follows:

nxproc = 4
nyproc = 2

A(4, 2)				
	A(1,1)	**A(2,1)**	**A(3,1)**	**A(4,1)**
	A(1,2)	**A(2,2)**	**A(3,2)**	**A(4,2)**

How MasPar Fortran arrays map on the machine

This type of mapping is frequently referred to as **cut and stack**.

The PE Grid

The PE grid on the MasPar is physically two-dimensional, although the PE local memory on each processor provides a virtual third dimension. Thus, 32-bit and 64-bit offsets in PE memory (PMEM) define virtual layers for REAL*4 and REAL*8 operands. These layers are analogous to vector strips on a vector machine. The analogy to strip mining on a vector machine is layering on a data-parallel machine. See Figure 2-1.

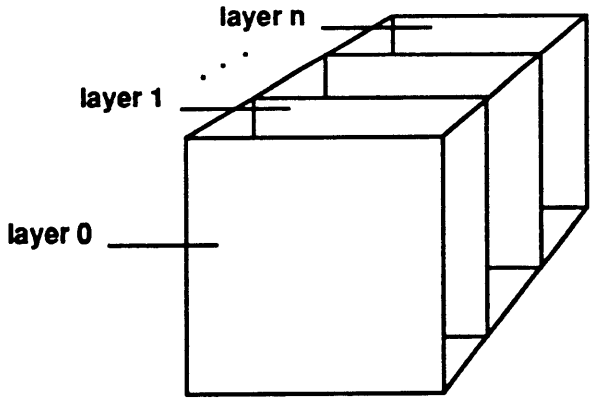


Figure 2-1 Virtual Layers into PE Memory

To explain the details of MasPar Fortran array allocation, a notation is used to specify where in PMEM a multidimensional array element is stored. Of primary interest is in which layer of a PE's local memory the array element (or section) is stored.

The following coordinate system is used in the examples in this section: the x and y coordinates of the PE in the grid specify its location. For example, [63,0] refers to the upper-right corner of a 4K machine (64 x 64 grid). Figure 2-2 illustrates the [x,y] PE indexing for a hypothetical 16-processor machine.

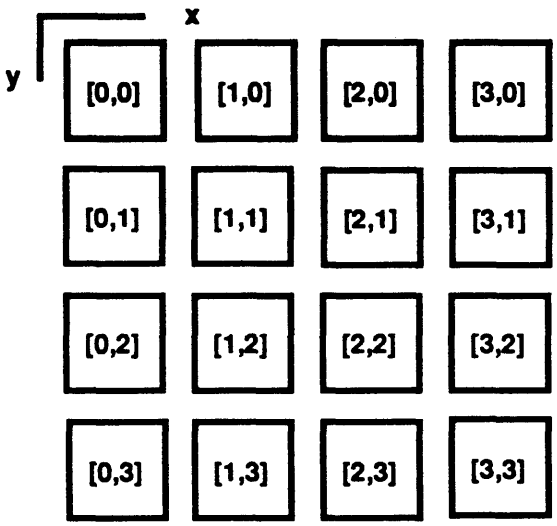


Figure 2-2 [x,y] PE Coordinates

The final coordinate is the layer in memory. An array that is smaller than (or equal to) the size of the PE grid fits into layer 0 of PMEM. Array sections that do not fit are spilled into layers 1, 2, etc., according to the MasPar Fortran array allocation algorithm. The location of an array element in PMEM is specified by the notation

[PE x coordinate, PE y coordinate, layer #]

In the coordinate notation, layer 0 of the upper-right corner of a 4K machine is represented by the triplet [63,0,0].

By declaring an array and using it in a parallel manner in the code, the compiler uses this default array mapping. You can force alternate mapping using the mapping directives. These are described on page 2-18.

One-dimensional arrays are mapped onto the PE grid in a **raster-scan** fashion. The first element of the array is mapped to [0,0,0]. Successive elements wrap around the grid from top row to bottom row. If there are more array elements than processors in the grid, wraparound proceeds to the next layer and the mapping is duplicated. Figure 2-3 illustrates the mapping for an array of length 8192 on a 4K machine (64 x 64 grid).

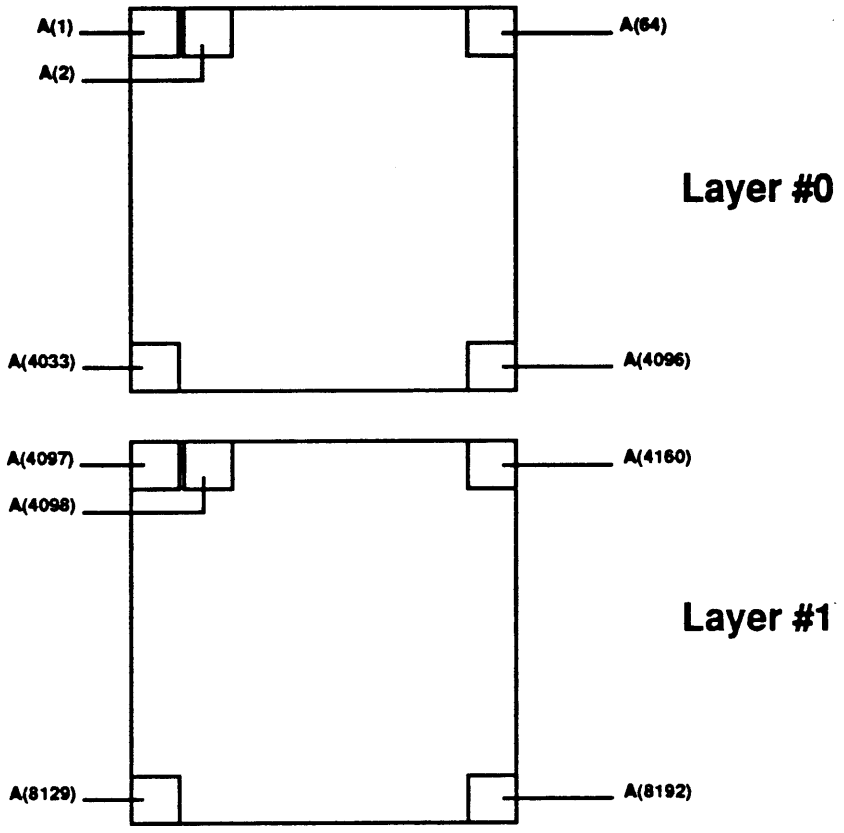


Figure 2-3 One-Dimensional Cut and Stack

Because Fortran stores column-major into linear FE memory, two-dimensional arrays are mapped column-to-row. Column one is mapped to row one of the PE grid. If the number of column elements (the first dimension) of the array exceeds the number of row elements of the PE grid, wraparound does not proceed to the next row of PEs. Instead it layers into memory. Column one of a two-dimensional array could thus easily occupy several layers in PMEM. Next, column two is mapped to row two of the PE grid and so on. If there are more columns in the array than rows of the PE grid, this creates more wraparound into memory.

Figure 2-4 illustrates the MasPar Fortran array mapping for the two-dimensional case for a 1K machine (32 x 32 grid) and an array A(50,50). While the layers in PMEM can be shown as stacked in a three-dimensional figure, they can also be laid out in a two-dimensional fashion. This particular example causes four layers in memory to be allocated. These layers are broken into quadrants, and the sections of A corresponding to each quadrant are shown.

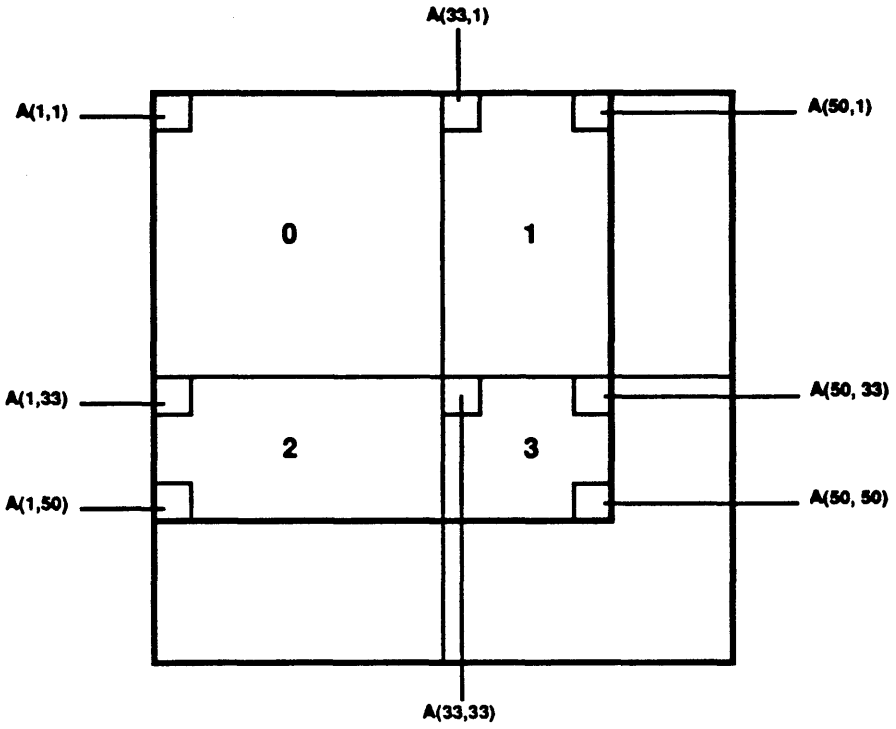


Figure 2-4 Cut and Stack for Two-Dimensional Arrays

By using this method of illustrating layering, we create a virtual machine. For two-dimensional arrays, the virtual machine size is determined by the following method. First, use the number of rows to determine what multiple of the target machine's x dimension is needed to entirely contain the row dimension of the array. Repeat for the number of columns and the target machine's y dimension. For example, how would an array of shape [225,9] be mapped onto a 4K machine (64 x 64 grid)? The multiple of the x dimension would be 4, since 225 <= 256 = 4*64. The multiple of the y dimension is 1, since 9 is already less than 64. The total number of layers is the number of x layers multiplied by the number of y layers. In this example, the four layers are laid out in the virtual machine, as shown in Figure 2-5. The thick solid lines are the boundaries of the virtual machine. Thin solid lines indicate how the virtual machine is composed of multiple layers of the original target machine. The dotted lines indicate the extent of the original array on the virtual machine.

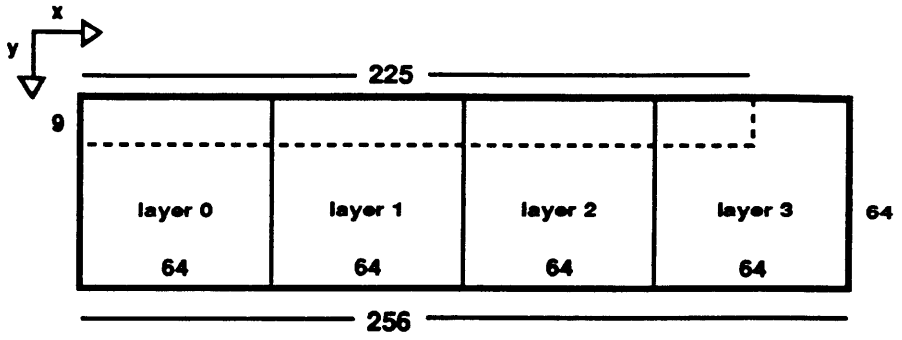


Figure 2-5 Mapping Array [225,9] on a 4K Machine

Three- and higher dimensional arrays are mapped to the PE grid using the cut and stack mapping for the first two dimensions and then layering successive planes into PMEM. In other words, A(1,1,1) and A(1,1,2) will have coordinates [0,0,0] and [0,0,offset], where *offset* is determined by the row and column dimension of A.

Figures 2-6 through 2-11 describe the array

real A(16, 16, 16)

mapped onto a 1K machine (32 x 32 grid). Each figure illustrates a three-dimensional view of the machine based on specific array statements. The shaded regions represent the array section in DPU memory referenced by the statement.

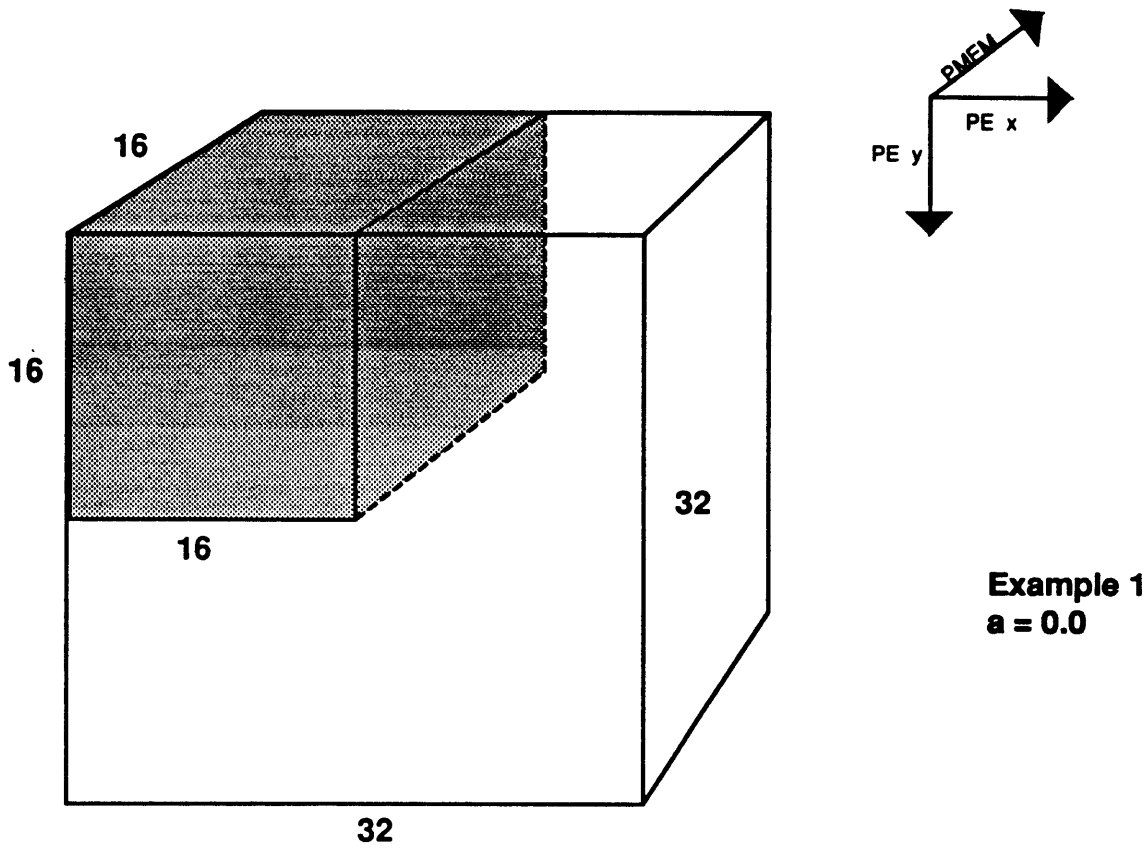


Figure 2-6 Example 1: a = 0.0

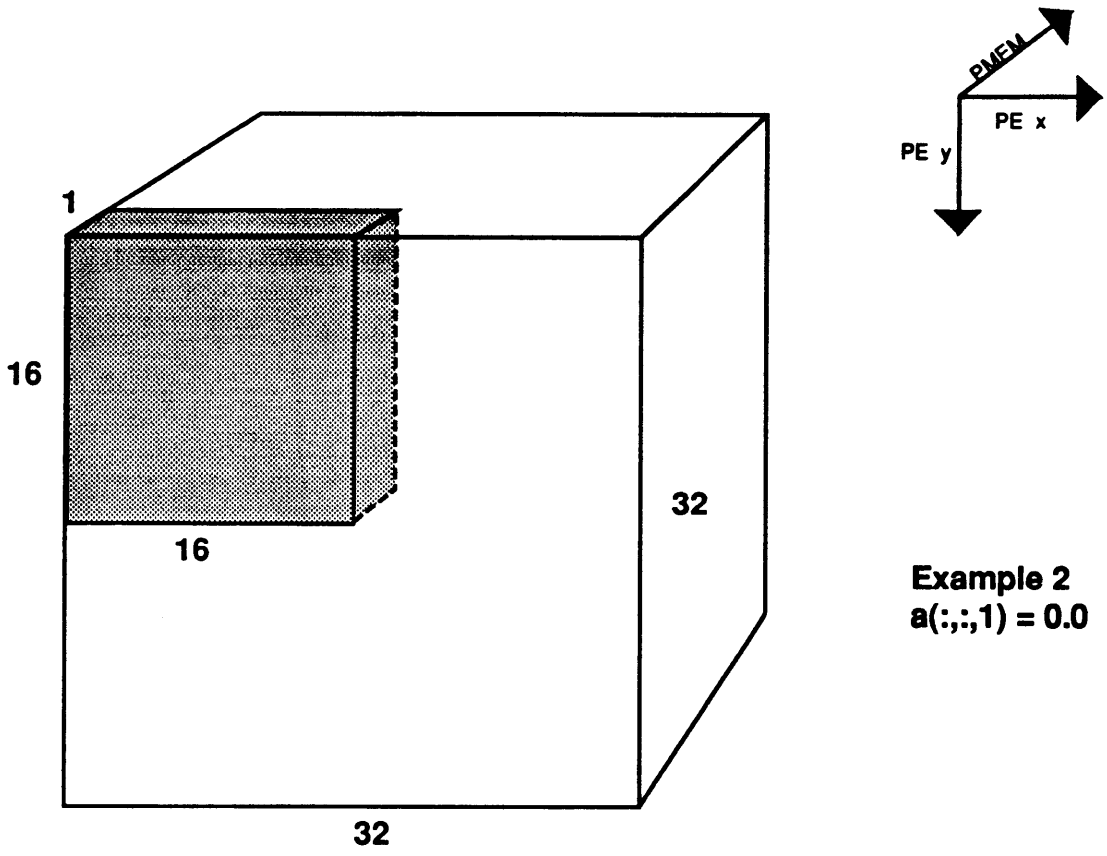


Figure 2-7 Example 2: a(:, :, 1) = 0.0

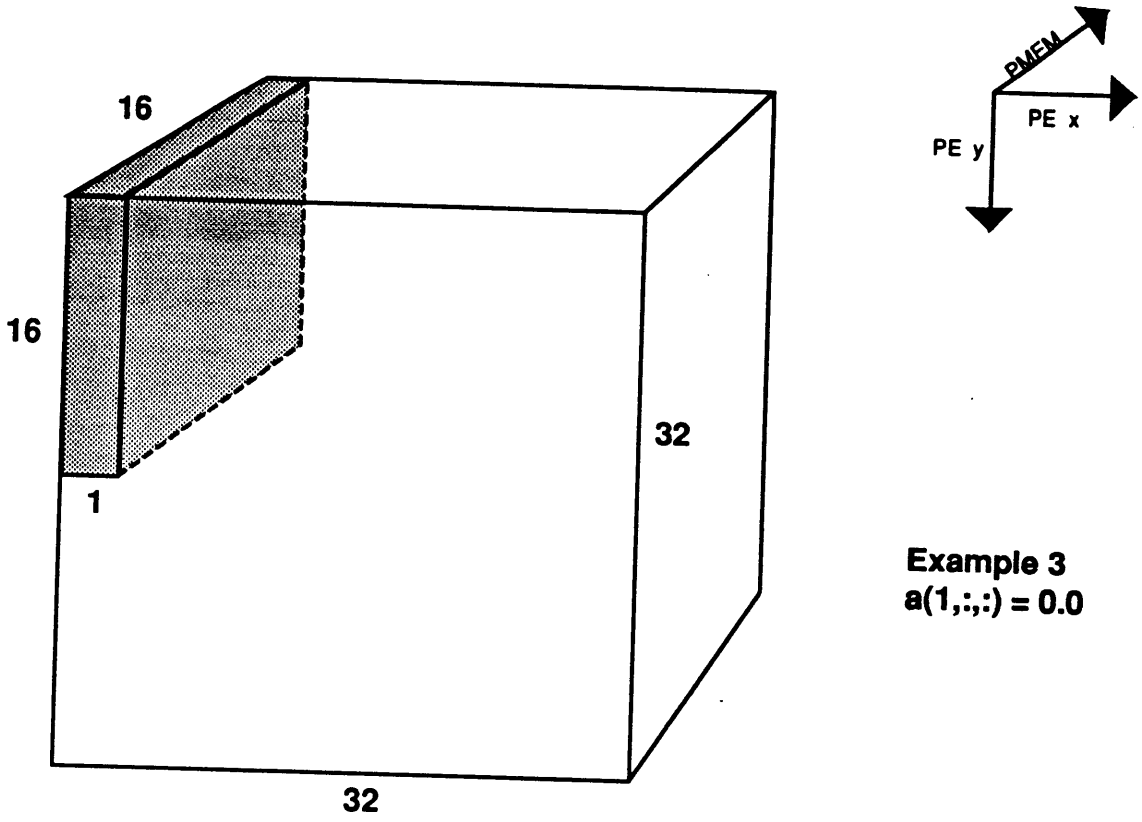
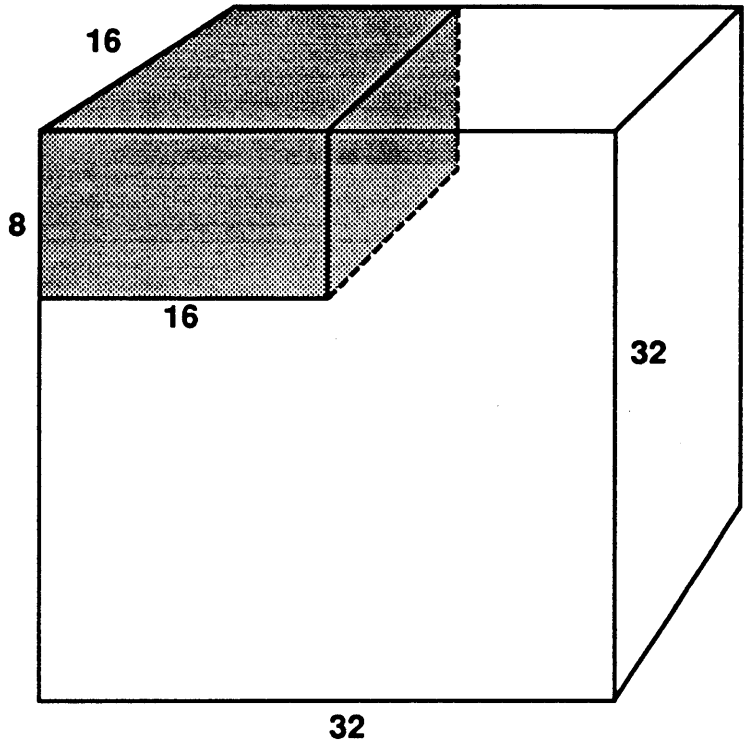
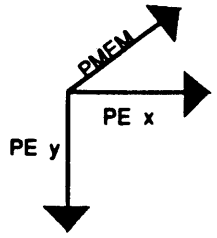


Figure 2-8 Example 3: a(1, :, :) = 0.0





Example 4
a(:,1:8,:) = 0.0

Figure 2-9 Example 4: a(:, 1:8, :) = 0.0

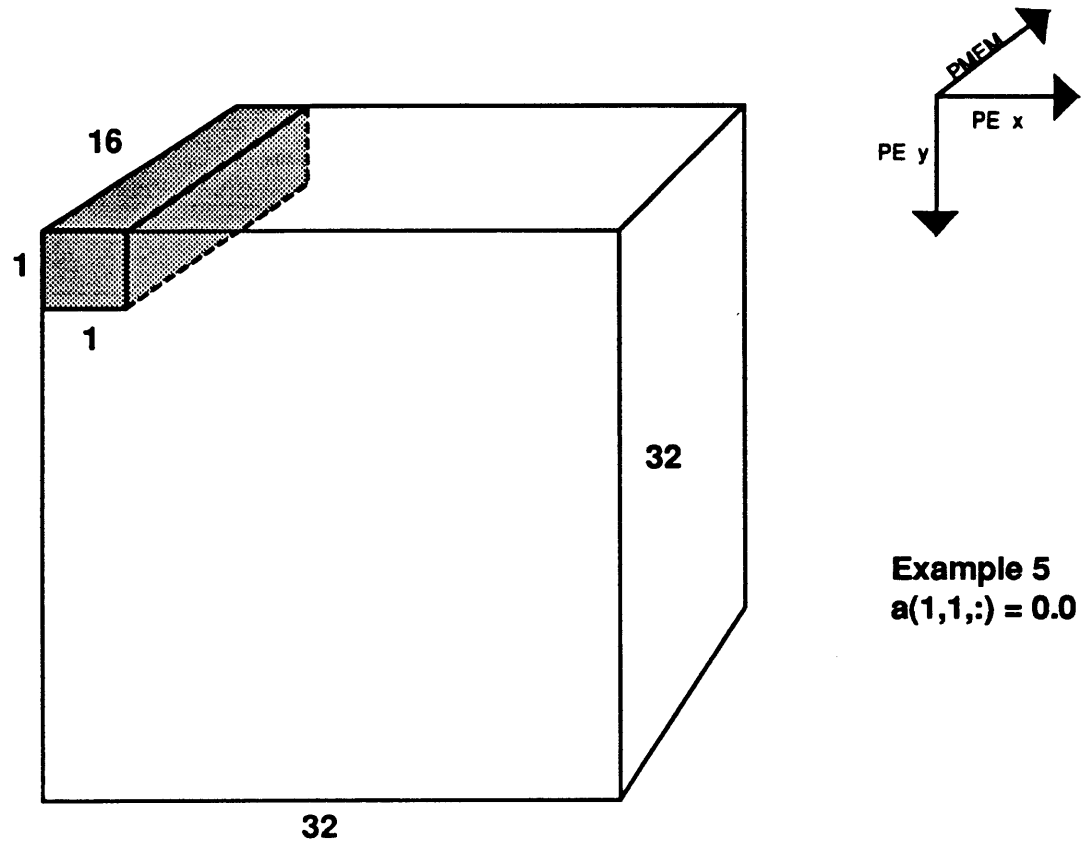
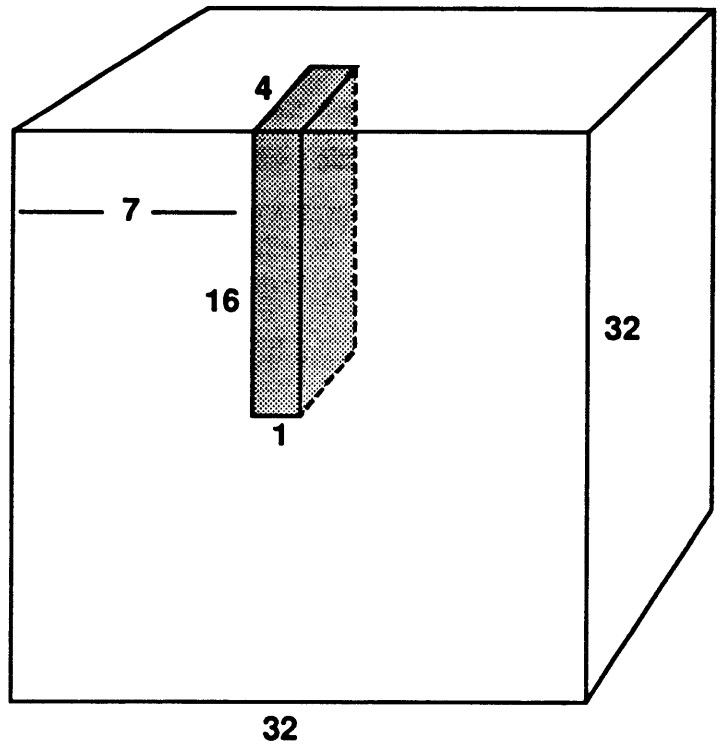
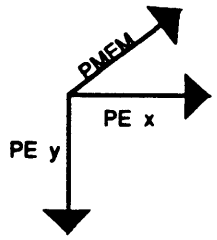


Figure 2-10 Example 5: a(1, 1, :) = 0.0





Example 6
a(8,:,1:4) = 0.0

Figure 2-11 Example 6: a(8, :, 1:4) = 0.0

Complex arrays are mapped onto the DPU in a manner similar to their real counterparts. The real and imaginary parts of each number are both stored on the PE corresponding to a particular array element.

So far, the array mapping has only been explained in the context of a single array. Allocation for all arrays, of *any* shape, begins at PE 0. Suppose you have three arrays A(100), B(100), and C(100). Then:

- A(1), B(1), and C(1) are stored at different offsets in the local memory of PE 0.
- A(100), B(100), and C(100) are stored at different offsets in the local memory of PE 99.

Mapping Directives

Mapping directives give you direct control over how arrays are allocated on the PE grid. Following are some examples of the mapping directives. (See the *MasPar Fortran Reference Manual* for details on syntax and usage of these directives.)

The following example causes all mapping directives that follow it in the same program unit to be disabled:

```
CMPF MAP
```

This example shows canonical (cut and stack) allocation:

```
real a(100,100,100,100)
CMPF MAP a(XBITS,YBITS,MEMORY,MEMORY)
```

In this example, the second dimension is allocated in the X dimension, and all other dimensions are allocated into memory:

```
real a(100,100,100,100)
CMPF MAP a(MEMORY,XBITS,MEMORY,MEMORY)
```

The following example allocates the first dimension to memory, then allocates the third dimension across all the processors, and then the second dimension across the rest of the processors. Whatever does not fit goes into memory.

```
real b(10,100,1000)
CMPF MAP b(MEMORY,PACKBITS:2,PACKBITS:1)
```

The following example of the bit specifiers allocates some of each of the three dimensions across the PE grid:

```
        dimension a(100,1000,100)
cmpf map a(0:1, 4:9, 2:3)
```

The first and third dimensions each use only 2 bits of addressing on the PEs, while the second dimension uses the remaining 6 bits of addressing on a 1K PE grid. In this way some of each dimension will be done in parallel. This gives the user the most control over how processors are allocated, including which specific PEs get which array elements. While this is more specific than most users usually need, many times this can result in the most efficient utilization of the PE grid.

The following example shows how dummy arguments and COMMON arrays can be used with mapping directives. By mapping the first dimension to ALLBITS and the second dimension to MEMORY, the PE grid of processors is much better used for arrays B and C. Notice that the mapping directives for the dummy array A and COMMON array B must match in both the called routine and interface block within every calling routine. INCLUDE files are a convenient way to ensure the directives will match.

```
          subroutine sub1(a)
          dimension a(:,:)
          common // B(10000,20)
cmpf map a(ALLBITS,MEMORY)
cmpf map b(ALLBITS,MEMORY)
                  .
                  .
                  .
          b(:,2) = b(:,2) + a(:,1)
                  .
                  .
          end

          program caller
          dimension c(10000,10)
cmpf map c(ALLBITS,MEMORY)
          interface
            subroutine sub1(a)
            dimension a(:,:)
cmpf map a(ALLBITS,MEMORY)
            end
          end interface
          common // B(10000,20)
cmpf map b(ALLBITS,MEMORY)
          c = 2
          a = 1
          call sub1(c)
          end
```

Note in this example that only the mapping directive for array C is optional. However, without it the call to sub1 would cause the array to be moved into the correct mapping twice, once before and once after the call, resulting in much poorer performance.

Look at Figures 2-12 through 2-15. These examples show four different ways to map the real array A(2048,128) onto a 4K machine. Note how different mappings of the same array can cause inefficient or very efficient use of the PE grid. The shading indicates which elements of the array A get mapped to the first memory layer.

REAL A(2048, 128)
on 4K machine (64x64)

Example 1: CANONICAL / XBITS, YBITS

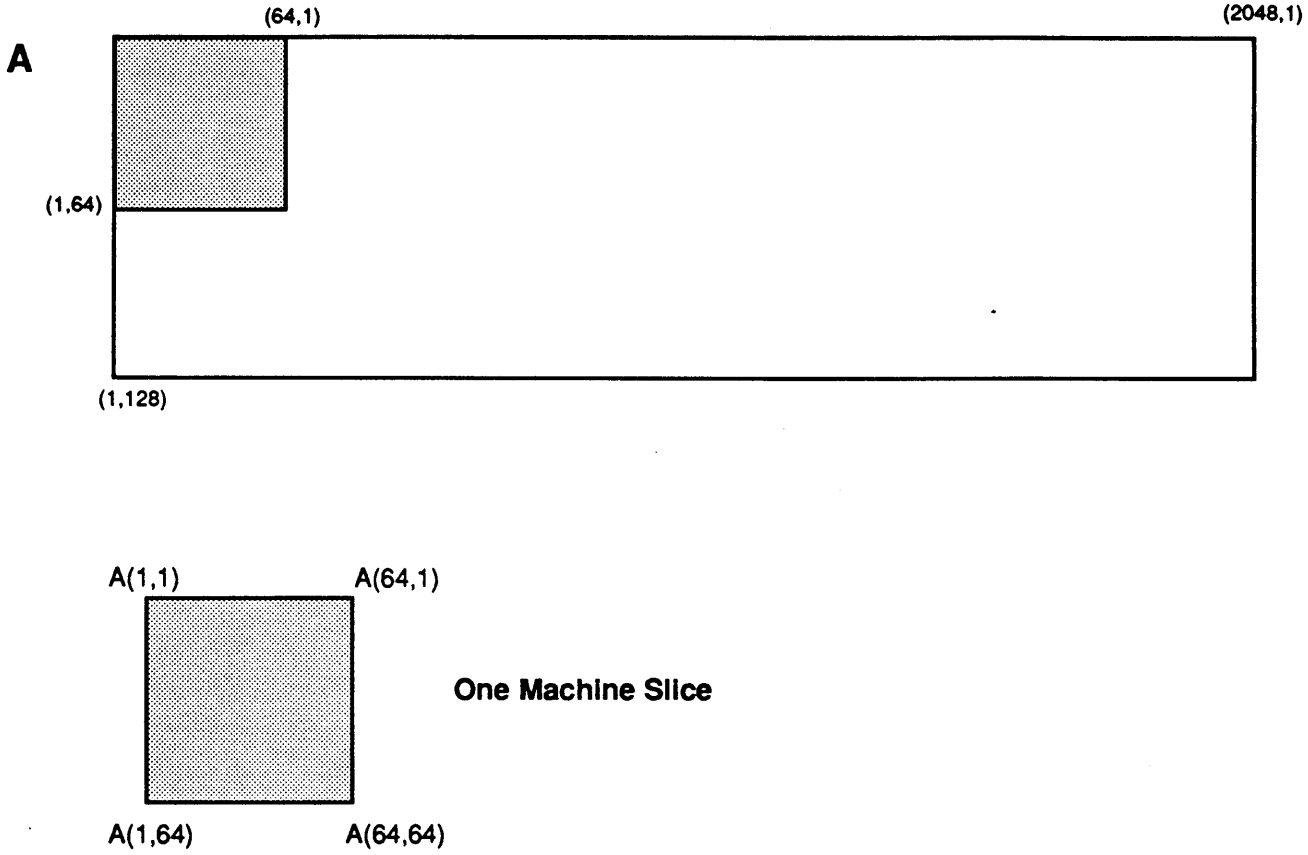
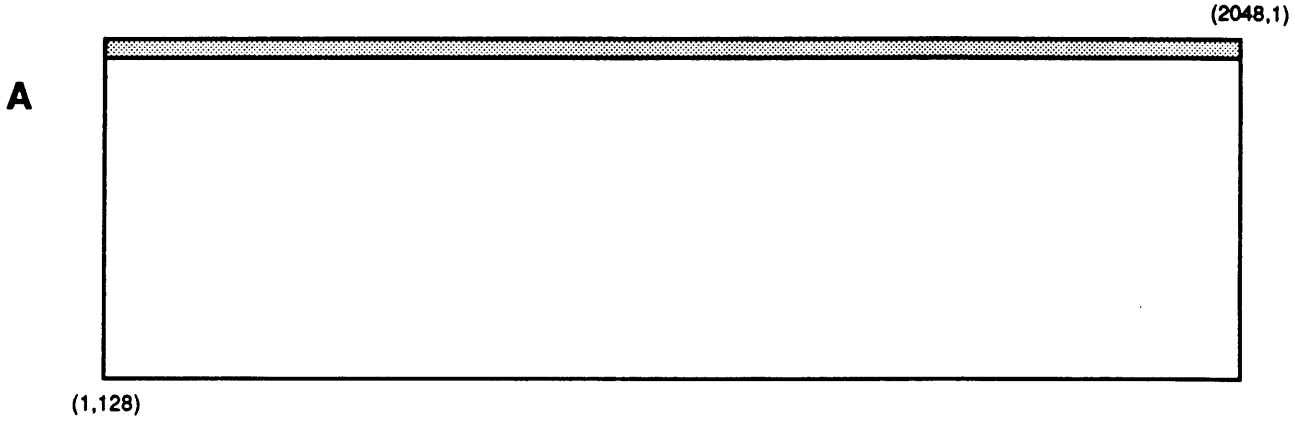
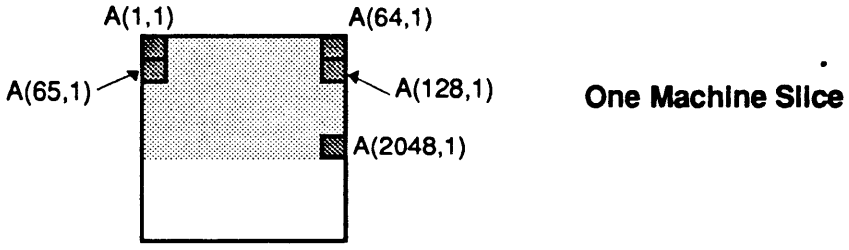


Figure 2-12 Mapping Example 1: Canonical / XBITS, YBITS

REAL A(2048, 128)
on 4K machine (64x64)

Example 2: ALLBITS, MEMORY



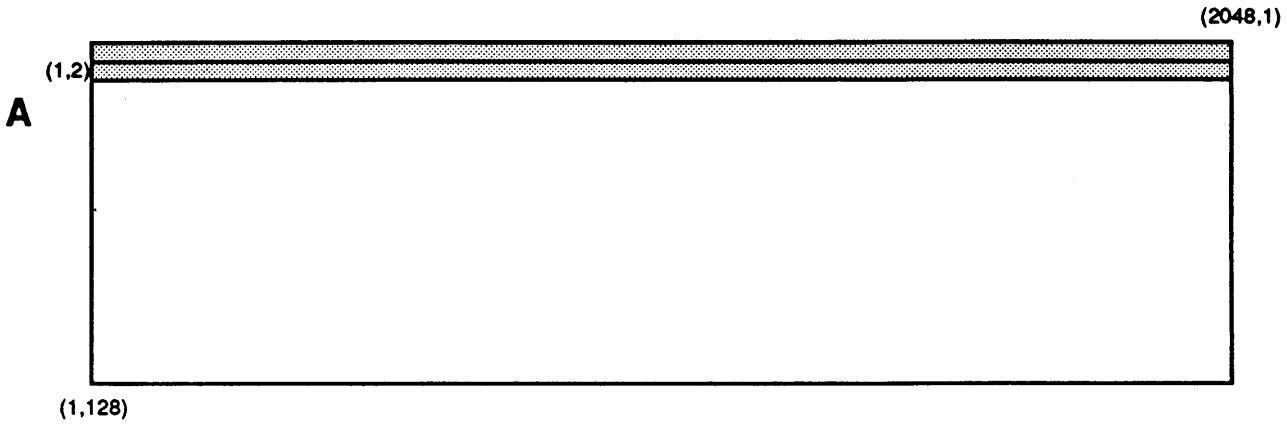


Note that only 2048 processors are used with this mapping

Figure 2-13 Mapping Example 2: ALLBITS, MEMORY

REAL A(2048, 128)
on 4K machine (64x64)

Example 3: PACKBITS:1, PACKBITS:2



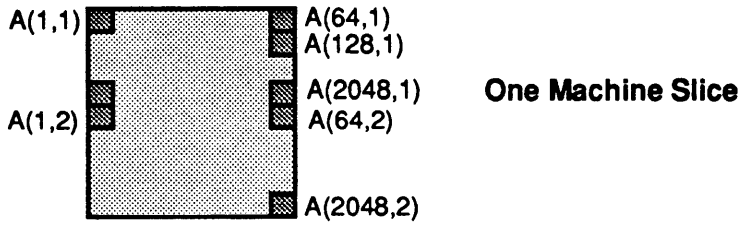
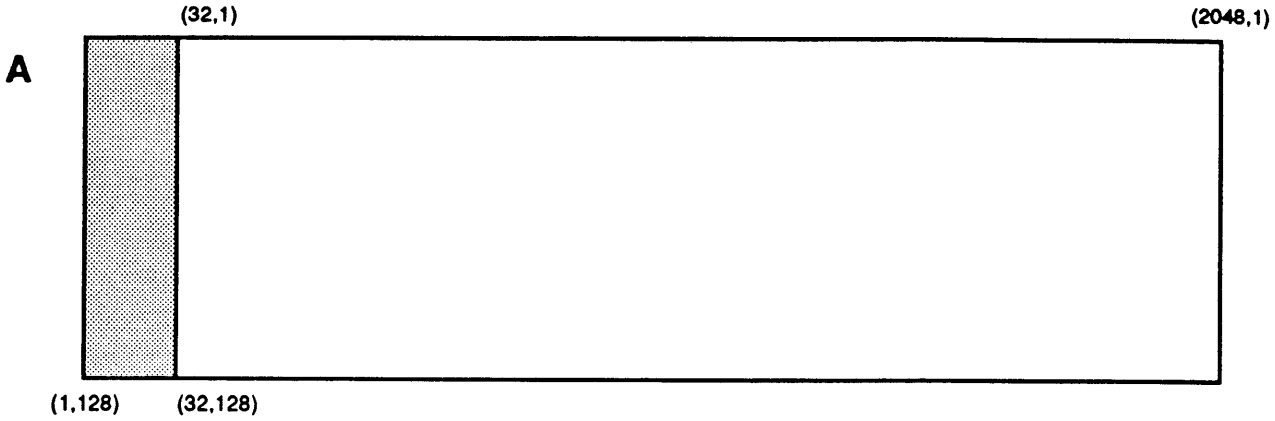


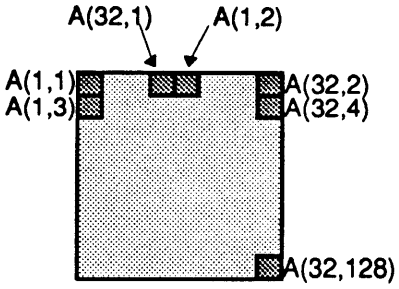
Figure 2-14 Mapping Example 3: PACKBITS:1, PACKBITS:2

REAL A(2048, 128)
on 4K machine (64x64)

Example 4: 0:4, 5:11

$2^5 = 32,\ 2^7 = 128$





One Machine Slice

Figure 2-15 Mapping Example 4: 0:4, 5:11

Array Alignment

Because communication is a major consideration in determining how fast code will execute, it plays a large role in array mapping. The cut and stack mapping method is efficient for the majority of data structures in a program. While MasPar Fortran could use a mapping that would exploit the entire extent of every dimension of an array, this might force router communication throughout the code. While the global router is efficient, it is far slower than X-Net communication. Therefore, a mapping that uses more PEs is not necessarily better if it imposes a greater communication cost.

In MasPar Fortran array allocation there is a noticeable difference between column-to-column and column-to-row operations.

In an intuitive sense, you can consider array alignment to refer to array operations in which the operands for each array operation have the same x and y coordinate. In this case, no PE-PE communication is required since the operands can be loaded directly from the PMEM of each individual processor. A simple array operation such as `c = a + b` is an example of perfect alignment, because this simply causes the values of `a` and `b` on each PE to be loaded, added together, and then stored in `c`.

A case of nonaligned arrays is one in which a PE needs an operand that is not in its local PMEM. If arrays cannot be aligned, the next best thing is to map them so that the communication occurs via the X-Net interconnect. For example, consider two arrays `A(N,N)` and `B(N,N)`, where columns 2 through n of `B` are replaced by the sum of themselves and columns 1 through n–1 of `A`. The Fortran 77 and Fortran 90 version of this code are shown here:

Fortran 77:

```
DO j = 2,n
   DO i = 1,n
      B(i,j) = B(i,j) + A(i,j-1)
   END DO
END DO
```

Fortran 90:

```
B(1:n,2:n) = B(1:n,2:n) + A(1:n,1:n-1)
```

In this case, the add cannot take place until the appropriate column elements of `A` have been transferred to the PEs containing the column elements of `B`. As it happens, the transfer is via X-Net, so array misalignment in this case is not terribly inefficient. In fact, it is very reasonable even in good data-parallel algorithms to have misaligned arrays. The most important performance consideration is to minimize the resulting communication cost. This example incurs a communication cost of an X-Net (distance 1) of rows 1 through n–1 of the array `A` (as stored on the PE grid). The X-net direction is south because of the array allocation on the DPU.

Since many matrix-oriented algorithms require column-to-column or row-to-row data transfer, the MasPar Fortran array allocation is an efficient algorithm, as most PE-PE communications will occur via X-Net.

Performance Ramifications

The key to good performance on a data-parallel machine is to use as many PEs as possible and to use them efficiently. On a 4K machine (64 x 64 grid), a 225 x 9 array will use at most 64*9 = 576 PEs, or just over 14% of the grid. Since dimensions three and higher are automatically layered into PMEM, the percentage of the PE grid used is determined by the row and column dimensions of the array with the *default* mapping. If one dimension is substantially larger than the other, then the grid will be used inefficiently. Use mapping directives to solve this problem.

What happens when increasing the problem size increases the mismatch between dimensions? For example, what happens if running a larger problem causes a 225 x 9 array to become a 450 x 9 array? You can control this by using MasPar Fortran mapping directives.

In another case an array is declared (16,256,128). Under the default mapping, the first two dimensions (16,256) determine the percentage of the grid used; this results in a very inefficient mapping. However, by using directives to map the last two array dimensions to the PE grid and the first array dimension to memory, you obtain a very efficient mapping.

You can also use the mapping directives to align a two-dimensional array with severely mismatched dimensions like a one-dimensional array. This is useful for the case where a 225 x 9 array becomes a 450 x 9 by increasing the problem size.

These are important concepts to understand. It is intuitive, but incorrect, to think that by multiplying the extent of each array dimension you can determine how many PEs are used. For example, an array A(225,9,2) does *not* use 4,050 PEs in a single array operation under the default mapping. Chapter 3 shows that the way arrays are accessed determines the types of communication used by the MPF compiler.

Under the default mapping, the number of PEs used in a rank-2 or higher rank array is determined by the shape of the first two dimensions. If you have a three-dimensional array, operations on each plane are done one at a time. This is how the compiler will map the arrays if you do not override the default by using mapping directives.

For example, consider the following arrays on a 4K machine:

```
REAL A(64,64,10), B(64,64,10)

B = B + A
```

While the code is a single-array operation, MasPar Fortran, in effect, translates this into the following code:

```
DO K = 1,10
```

```
            B(:,:,K) = B(:,:,K) + A(:,:,K)
END DO
```

At each iteration of the K loop, another plane of A and B is loaded into PMEM, the add is performed, and the result is stored.

Chapter 3

Communication

Chapter 2 showed how PE-PE communication could occur due to misalignment of arrays. In MPL, explicit control and virtualization of the PE grid is required of the programmer. Because of this, data placement and communication are entirely at the user's discretion. The user has the freedom to switch dynamically from one type of communication to another or to choose a particular form of communication based on problem size.

In contrast, the MPF compiler determines the communication methods to use based on the way the code is written; therefore the programmer "controls" communication by using effective programming techniques. Programming style and array access have a direct effect on the type of communication used. Therefore it is important to understand how MasPar Fortran makes communication decisions and what effect you can have over it.

MasPar Fortran does not switch dynamically from one type of PE-PE communication to another. In other words, X-Net or global router is not selected based on a runtime test of problem size. Therefore, when PE-PE communication is required, the type of communication used is determined by how the arrays are aligned. For details on these forms of communication, see the *MPL Manuals* and the *MasPar Architecture Specification*. Additionally, because MasPar Fortran manages the front end/DPU interface for you, it is useful to understand front end/DPU communication and when this will occur in your code. This chapter highlights certain communication guidelines that directly affect the level of performance obtained from your code.

Front End-to-DPU Communication

The most important distinction that MasPar Fortran makes is between scalars and arrays. In Fortran 90, multidimensional arrays are treated as first-class language objects. They are defined, subselected, and manipulated as whole objects. However, these objects can be broken down into smaller and smaller objects that reduce to a collection of scalar elements.

In MasPar Fortran there are two types of "scalars". The first is the familiar, single variable such as X, PI2, etc. The second is an array used in a scalar context. Scalars are defined in statements such as X = 2.5, and so forth. However, consider an array

```
REAL A(100)
```

that can be used in two contexts:

```
do i = 1,100
   A(i) = 1.0          ! Fortran 77 scalar context
end do

A = 1.0                ! Fortran 90 array context
```

In the first usage, the array is accessed one scalar element at a time. As far as MasPar Fortran is concerned, this is a case of using an array (a valid MasPar Fortran object) in a scalar context. The array elements are assigned one element at a time, just as in a scalar machine. However, in the second usage of A (an array used as an object in a parallel context), the entire object is initialized in one instruction. The operation is carried out in parallel.

NOTE: In MasPar Fortran, scalars and arrays are distinct objects. They can be operated on by separate processors.

In MasPar Fortran, scalars are operated on by the front end. Arrays used in a parallel context are operated on by the DPU.

- *Front-End Memory and Front-End Operation*:
  - Scalars (traditional, single elements)
  - Arrays used in a scalar context
- *DPU Memory and DPU Operations*:
  - Arrays used in a parallel context (Fortran 90 array syntax)

An important consequence of this distinction is that while you can run standard Fortran 77 code under MasPar Fortran, it will execute in scalar on the front end.

In the preceding example with the array A, the two operations shown can be carried out and stored on two entirely different machines. In addition to the cost incurred by the computation (parallel operations happen faster than sequential), a communication cost might also be incurred. Suppose we changed the preceding example slightly to the following:

```
real A(100), B(100)
```

```
A = 1.0

do i = 1,100
   B(i) = 2.0
end do

B = A + B
```

The array statement A = 1.0 causes A to be allocated in DPU memory and assigned the value of 1.0 in parallel. Since B is used in an array context, it is also allocated in DPU memory. However, the serial loop causes the elements of B to be stored into PE memory (PMEM) one element at a time. Note that arrays can be used in a scalar context even on the DPU. If the array is already current in DPU memory, then scalar usage (i.e., Fortran 77 loops) causes context to be set to individual PEs, one at a time.

The next array operation is performed in parallel. If either A or B is written to a disk file on the front end, data movement will occur from DPU memory to the front end to make the data current on the front end.

Every time an array is used in a scalar context inside a loop, the elements are accessed one at a time. If additional movement is required to transfer the data from front end memory to DPU memory, then even more overhead is required. This has a noticeable effect on performance. However, if a scalar is needed for a scalar-array operation, the broadcast is handled in two steps. The scalar is first transferred to the ACU and then broadcast to the PE grid. The broadcast is quite fast. Code of the following form

```
real A(100), B(100), C

B = C * A
```

causes the value of C to be transferred from front-end memory to the ACU and then broadcast to the PE grid. In effect, two communication steps are required. Conversely, the same two steps happen in reverse in the following code:

```
B = ...
   .
   .
   .
C = B(10,10)
```

In this case, after B is computed, the value of B(10,10) is transferred from the appropriate element in PMEM to an ACU register. It is then transferred from the ACU to front-end memory.

PE-PE Communication

As noted earlier, regular column-to-column or row-to-row operations are translated into X-Net instructions by MasPar Fortran. This is the primary benefit of the MasPar Fortran array mapping. However, an operation that accesses both a column of one matrix and a row of another is quite different. Consider the following loop:

```
real A(16,16), B(16,16)

do i = 1,16
   B(i,2) = B(i,2) + A(2,i)
end do
```

The second column of B is updated by adding the second row of A to it. Figure 3-1 illustrates the appropriate column and row, according to MasPar Fortran array allocation.

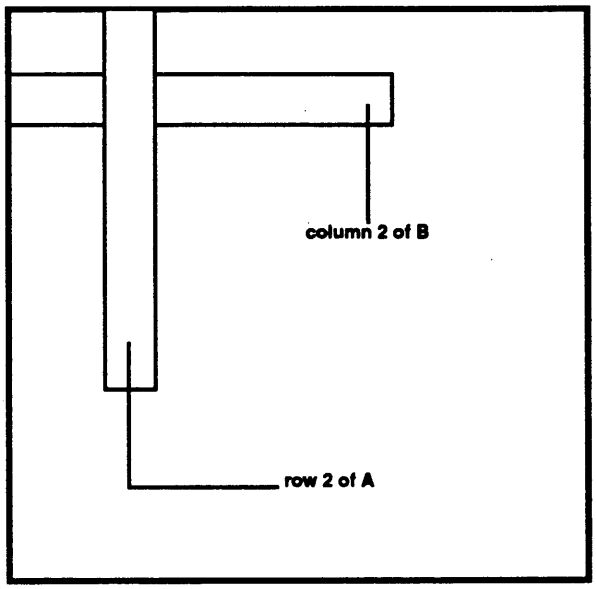


Figure 3-1 Two-Dimensional Array Misalignment

To align the row elements of A with the appropriate column elements of B, both X-Net communication and global router operations are required. For more information on PE-PE communication, see the *MasPar Architecture Specification* and the *MPL Programming Manuals*. Remember too that by using the mapping directives, you can force alternate mapping (see page 2-18).

Chapter 4

Developing MasPar Fortran Programs

MasPar Fortran is a data-parallel implementation of Fortran 77 and the Fortran 90 array processing extensions.

In Fortran 90 arrays are treated as single objects. The most distinctive difference between a conventional (or serial) implementation of Fortran 90 and a data-parallel implementation is that in a serial implementation the compiler will perform iterative loops to process each element in an array object. By contrast, the MasPar system is a data-parallel processor. In MasPar Fortran each element of a Fortran 90 array object is stored in its own processor element (PE), so all the elements of an array object can be operated on simultaneously.

This chapter provides tips on how to write effective MasPar Fortran programs. It assumes you are familiar with programming in Fortran 77 and that you have some knowledge of parallel programming concepts.

Programming Tips

There are many aspects to writing effective MasPar Fortran programs. Some techniques and constructs that were effective in Fortran 77 programming are not necessarily well suited to a data-parallel machine. Some of these are noted here.

Arrays and Indexed Array Elements

In Fortran 77, array elements are often passed as if they were actual array addresses. In MasPar Fortran, this is not a problem as long as both the calling routine and the called routine are Fortran 77 routines. Problems arise, however, when the Fortran 77 array element is mixed with a call to a Fortran 90 subroutine expecting an array rather than a scalar array element.

The following program shows an illegal Fortran 77 scalar array element mixed with a call to a Fortran 90 subroutine expecting an array:

```
program example2
integer, parameter :: M = 100, N = 100
integer A(M, N)

  ...
  call foo( A(1, j), M, N-j+1 )     ! This usage is ILLEGAL
end


subroutine foo( A, M, N )           ! A Fortran 90 subroutine
integer M, N
integer A( M, N )

  A = A + 1
end
```

The above illegal program can be corrected by recoding so that the array is passed as shown below. Sectioning is used to have foo operate only on the part of the array starting at (i, j).

```
program example2
integer, parameter :: M = 100, N = 100
integer A(M, N)

  ...
  call foo( A(1:M, j:N) )           ! Array section rather than
end                                 ! scalar array element


subroutine foo( A )
integer A( :, : )

  A = A + 1
end
```

When you do not want to modify a Fortran 77-style main program that passes a scalar array element as if it were a section to a Fortran 90-style subroutine, you can use the following "explicit copy subroutine" technique. Bear in mind that the array copying from the front end to/from the DPU might cause a performance problem and main program modification might be necessary.

```
program test
implicit none
integer, parameter :: size = 20, lda = size, n = 18, k = 2
integer, dimension (size, size) :: a

  call foo( a(k, k), n, lda )       ! 77-style section by passing
end                                 ! scalar array element

subroutine foo( x, n, lda )
implicit none
integer n, lda
integer, dimension( lda, n ) :: x   ! keep x on the front end
integer, dimension( n, n ) :: b     ! by never using it in a
                                    ! Fortran 90-style statement

  call copyin(x, b, n)
                                    ! Use "b" in 90 style here
  call copyout(x, b, n)
end


subroutine copyin(x, b, n)
implicit none
integer n
integer, dimension( :, : ) :: x
integer, dimension(n, n) :: b

  b = x(1:n, 1:n)
end


subroutine copyout(x, b, n)
integer, dimension( :,: ) :: x
integer, dimension(n, n) :: b

  x(1:n, 1:n) = b
end
```

Aliasing

Certain forms of aliasing across subroutine boundaries are undefined in Fortran 77 and cause different behavior when compiled with different compilers on different computer systems. For example, in the code below the variable X is used twice as an argument to subroutine wrong. Within the subroutine, the arguments A and B reference the same memory location, which may cause unexpected results.

```
integer X

call wrong( X, X )
end

subroutine wrong( A, B )
integer A, B

  A = 3
  B = A * 2
  A = A + 4
end
```

A similar problem can arise in MasPar Fortran. In the code below, the call to subroutine a_k_a creates an aliasing problem with a(7,7) and the array section a(start:fin, start:fin). When it compiles the subroutine call

```
call a_k_a( a(7, 7), a(start:fin, start:fin) )
```

the compiler generates code for the following:

```
temp = A(7, 7)
temp2 = a(start:fin, start:fin)
call a_k_a( temp, temp2 )
a(start:fin, start:fin) = temp2
A(7, 7) = temp
```

Note that temp is a temporary on the front end, and temp2 is a PE-based temporary.

```
      program alias
      implicit none
      integer, parameter :: size = 10
      integer, dimension( size, size ) :: a
      integer i, j, start, fin

        call init( a, size )
        start = size/2
        fin = size
        call a_k_a( a(7, 7), a(start:fin, start:fin) )
        do j = 1, size
           print 100, (a(i, j), i = 1, size)
100        format ( 1X, 10( I4 ))
        end do
      end

      subroutine a_k_a( val, section )
      integer val
      integer, dimension( :, : ) :: section

        section = 0
      end
```

```
! Number all the elements in the array "a"
subroutine init( a, size )
integer size
integer, dimension( size, size ) :: a

  do j = 1, size
     do i = 1, size
        a(i, j) = i + (j-1) * size
     end do
  end do
end
```

When this program is executed, it prints the following matrix:

```
 1   2   3   4   5   6   7   8   9  10
11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  30
31  32  33  34  35  36  37  38  39  40
41  42  43  44   0   0   0   0   0   0
51  52  53  54   0   0   0   0   0   0
61  62  63  64   0   0  67   0   0   0
71  72  73  74   0   0   0   0   0   0
81  82  83  84   0   0   0   0   0   0
91  92  93  94   0   0   0   0   0   0
```

Note that the element A(7,7) is unchanged by the assignment of the section to zero. This is because the temporary for the scalar argument A(7,7) is copied back into the array *after* the array section temporary is copied back into A. Since the temporary for A(7,7) retains the original value, this element remains unchanged in A.

Argument Copying on a Subroutine Call

Because the front-end memory and the DPU memory are disjoint, the subroutine call interface is more complex than it would be for a Fortran compiler on a machine with an unbroken linear memory. For example, the code below passes an indexed array element as an argument to the subroutine foo:

```
call foo( A(I) )
```

If A is on the front end, the address of A(I) is passed to the subroutine. However, if A is on the DPU, the situation is more complex. A(I) is a scalar reference, so the argument A(I) must be on the front end. This means that the DPU value A(I) must be selected on the PE array and copied to a front-end temporary. This temporary is then used as the argument. To preserve the semantics of Fortran argument passing, the temporary is copied back into A(I) after the call. This calling sequence is shown below:

```
temp = A(I)
call foo( temp )
A(I) = temp
```

Data movement also occurs when an array section is passed as a subroutine argument. For example:

```
call bar( B(I:J:K) )
```

Because array B is used in an array expression (it is sectioned), it resides on the DPU. Before the subroutine call, the array section B(I:J:K) is moved (via the global router) into a DPU-resident temporary. This temporary is used as the argument in the subroutine call. After the subroutine call, the temporary is copied back into the original array, B. This calling sequence is shown here:

```
temp = B(I:J:K)
call bar( temp )
B(I:J:K) = temp
```

Storage Management

A common technique used in Fortran 77 programming for storage management is to declare a huge array of storage, then pass parts of it (as different subarrays and temporary storage) to the routines doing the calculation. This is not necessary in MasPar Fortran and it is not recommended. This method limits the global size of the problem that can be handled and wastes storage, because the maximum size must be declared even when it is not used.

For arrays used in a Fortran 90 manner, use automatic arrays in subprograms to make efficient use of storage. The compiler can generate more efficient code if local storage is used rather than COMMON storage or globally managed arrays. By using automatic arrays you avoid the overhead of passing temporary work arrays.

During the early stages of porting a program to MasPar Fortran, your main program might have large arrays that are not used in a Fortran 90 manner. Place these in COMMON to avoid slow linking and a large object file. By the end of the porting process, be sure all large arrays are handled with Fortran 90 operations. (See Chapter 5 for information on converting Fortran 77 programs.)

Do not use EQUIVALENCE of arrays to try to save storage; this makes it more difficult for the compiler to perform data optimizations and causes much less efficient code to be generated.

One-dimensional arrays are spread across the PE grid. For arrays of rank greater than one, only the first two dimensions of an array are allocated across the PE grid by default. The remaining dimensions are put into memory. Therefore, make the first two dimensions of your array the largest dimensions, thus ensuring their placement on the PE grid, and more efficient use of available storage. Alternatively, you can use the mapping directives to force alternate placement. See the discussions of the mapping directives in Chapter 2 of this user guide and in the *MasPar Fortran Reference Manual.*

Compile-Time Analysis

To produce tuned, high-performance MasPar Fortran code, you need to know how your problem maps onto the system and what machine resources are being used. Program analysis tools are available at the mpfortran command-line, with the command-line program mpprof, and with MPPE. These tools give you useful information about your program's data mapping, communications, and data movement. Analyzing this data helps you understand which areas of your program are causing performance bottlenecks, and what constructs you can change to improve performance.

You can generate compile-time analysis information at the time you compile your program using mpfortran command-line options. This information is useful for tuning individual files as you port your program to MasPar Fortran.

To display the compile-time messages on screen as you are compiling, use the following mpfortran command-line options.

–report	Prints a report of compile-time analysis messages for each subprogram on standard output. If the –V option is specified, the messages will also be included in a listing file. Note that the following 3 options (–threshold=*n*, –blocked, and –storage) all force the –report option.
–threshold=*n*	Forces the –report option, printing only compile-time analysis messages with a level less than or equal to *n*. The default is *n*=0. As a general rule, as the level increases, the messages become less critical and might be more voluminous. The highest level is 2; all possible messages are generated when –threshold=2. Note that the following 2 options (–blocked and –storage) ignore the –threshold option.
–blocked	Forces the –report option, printing only compile-time analysis messages that indicate multilayer array mappings that might affect blocked algorithms, regardless of the –threshold setting. (See page 4-14.)
–storage	Forces the –report option, printing only compile-time analysis messages that indicate storage usage, regardless of –threshold setting. (See page 4-14.)

To create a printed report of these messages, use the –V option with any of the above options; this will generate a .lis file that has the compile-time messages collated into the source listing. See Chapter 2 of the *Commands Reference Manual* for details about the reports that are generated.

The compile-time information available through the mpfortran command-line is also available with MPPE and mpprof. MPPE provides a graphical interface, and mpprof provides a command-line interface. Both tools can be used to generate reports.

Compile-time profiling shows the *estimated* amount of resources your MasPar Fortran program will use, based on an analysis of the code performed at compile time. The analysis of the program is based on an internal system of weights. The weights are arbitrary numbers assigned to various time-intensive operations; higher numbers are assigned to operations that typically require more time. The compile-time profile weights for typical programs come from data communications operations, including data movement between the front end and the DPU, and PE communications (global router operations and X-Net operations). See the *Commands Reference Manual* and the *MPPE User Guide* for more information on using mpprof and MPPE, respectively.

Each compile-time analysis message is assigned a unique identification number. The following list identifies the message numbers associated with each topic:

Messages related to program performance:

Message #21	Array sloshing
Messages #201 through #205	Router operations
Messages #301 through #305	X-Net operations
Messages #391 through #392	Scalar sloshing
Messages #551 through #552	DPU array element access
Messages #602 through #620	Scalar FORALL processing
Message #801	Scalar array I/O
Messages #901 through #902	Strip mining
Message #1001	Loop fusion
Messages #1251 and #1253	Blocked algorithm inhibition

Messages related to use of storage space:

Messages #1801 through #1829	Storage

All these messages can be generated on the mpfortran command-line, with mpprof, and with MPPE, except for message #21, which is not available at the mpfortran command line. To generate messages 902 through 1829 with mpfortran, additional command-line options are required (–threshold, –blocked, or –storage).

Array Sloshing between Front End and DPU Arguments

Moving data between the front end and the DPU, or "sloshing", can cause significant performance degradation. This movement happens when an array on the front end is passed to a routine that has that array on the DPU, or when an array on the DPU is passed to a routine that has that array on the front end. Several factors determine whether an array is on the front end or the DPU in a given routine. Use of an ONFE or ONDPU directive can determine an array's location. Lacking a directive, a COMMON array appears on the front end by default, or on the DPU if a –nofecommon appears on the mpfortran command line. Lacking a directive, a non-COMMON array appears on the DPU if it is used in a Fortran 90 context (*not* by declaration, however) and on the front end otherwise. (See Chapter 2 for more information on data allocation.) Array sloshing can be detected with mpprof or MPPE (not on the mpfortran command-line); it is reported with Message #21.

Following is a sample program that has a sloshing problem:

```
program foo
real a(100)
a = 1
call sub1(a)            ! see first line of message #21
end

subroutine sub1(a)
real a(100), b(200)
b = 1
call sub2(a, b)         ! see second line of message #21
end

subroutine sub2(a, b)
real a(100), b(200)
a(1) = 1                ! see third line of message #21
b(1) = 1
end
```

This program will generate the following warning message:

#21 **array slosh from DPU arg 1 in FOO(t5.f:4)**
thru arg A in SUB1(t5.f:10)
to FE arg A in SUB2()

In this example, an array starts in FOO, is passed through SUB1 without being used (except as an argument), and arrives in SUB2. The first line of message #21 describes where it started in FOO. The second line describes its passage through SUB1. The third line describes its arrival at SUB2. In general, because an array can be passed through any number of intermediate routines that do not use it, this "slosh" message can have any number of lines like the second line in this example.

You can eliminate the sloshing by adding an ONDPU mapping directive for A in SUB2, or by passing a front-end array as ARG1 at FOO(t5.f:4).

Global Router Operations

Global router operations result from irregular communication patterns; this type of communication is efficient but can be 16 times slower than X-Net communication. Therefore, router operations can cause performance degradation. Router code can be generated, for example, by vector-valued subscripts. (See Chapter 3 for more information on communication.)

Many times router operations can be avoided by recoding or modifying the source program. For example, instead of vector-valued subscripts, you might be able to use the table lookup feature of the FORALL statement (see page 4-18). Also, using mapping directives will help you avoid router communication in some cases. Mapping directives might apply, for instance, if you have mismatched array sections, as in A(1,:)=B(:). Following are the compile-time analysis messages that relate to router operations:

#201 **router send**

#202 **router fetch**

#203 **router store**

#204 **router indirect load**

#205 **router indirect store**

X-Net Operations

X-Net operations result from regular communication patterns (for example, CSHIFTs and EOSHIFTs). The impact on performance varies, depending on the distance of the X-Net movement. X-Net is the fastest type of communication, preferable to router communication. (See Chapter 3 for more information on communication.) Following are the compile-time analysis messages that identify X-Net operations:

#301 **xnet constant distance**
This is the fastest type of communication. The only better situation is no communication at all.

#302 **xnet variable distance**
This type of communication is less efficient than compile-time distance X-Net communication. Check your program to see if the variable shift distance can be replaced by a constant shift distance.

#303 **xnet constant distance raster shift**
This could be caused by shifting one-dimensional arrays a constant distance.

#304 **xnet via subroutine call**
X-Net communication is being used; however, the dimension being shifted is not mapped to align with the machine X-Net grid.

#305 **spread via subroutine call**
A SPREAD operation might use X-Net communication to create the additional dimension, depending on the SPREAD dimension and how the input and output are mapped.

Scalar Sloshing between the Front End and the DPU

Sloshing is data movement between the front end and the DPU. (See Chapter 2 for more information.) Messages #391 and #392 identify when scalar data is moved back and forth, slowing performance.

#391 **scalar slosh to the DPU**
Scalar data is moving from the front end to the DPU.

#392 **scalar slosh from the DPU**
Scalar data movement from the DPU to the front end.

Using the –Omax command-line option will frequently reduce the number of scalar sloshes. However, there are cases where you can change your code to reduce the number of scalar sloshes even further. Consider the following example, where "m" and "n" are used as array indices.

```
subroutine index(m, n)
real, array(100, 100) :: a, b, c
a(1:m, 1:n) = b(1:m, 1:n)
b(1:m, 1:n) = c(1:m, 1:n)
end
```

When compiled, this routine gets 10 scalar sloshes without –Omax, and 3 scalar sloshes with –Omax. However, if the code is changed to replace the variables by parameters, it gets no scalar slosh at all! Such a change can substantially improve performance. Following is the revised code:

```
subroutine index(m, n)
parameter(i = 90, j = 95)
real, array(100, 100) :: a, b, c
a(1:i, 1:j) = b(1:i, 1:j)
b(1:i, 1:j) = c(1:i, 1:j)
end
```

Although this type of change will not always apply, it might be possible when:

- the variables are actually parameters or constants in the calling program
- an operation can be done on a larger, constant-sized section of an array

DPU Array Element Access

If an array that is stored in DPU memory is accessed in a scalar manner, performance can be affected.

#551 **DPU array single-element fetch**
For example:

```
real, array(100,100) :: A
real :: const
:
:
const = A(2,5)
```

#552 **DPU array single-element store**
For example:

```
real, array(100,100) :: A
real :: const
:
:
A(2,5) = const
```

When these messages appear in DO loops there are two alternatives:

- If at all possible, vectorize the statement.
- If the statement cannot be vectorized but is a very large array operation, it will be faster to have the array ONFE.

Scalar FORALL Processing

Because of the generality of the FORALL statement and an incomplete implementation in MasPar Fortran, several factors can currently inhibit parallelization. Future releases of the compiler will remove most of these. Executing a FORALL statement serially can impact performance significantly. Messages #602 through #620 identify some of reasons that the FORALL statement will be executed serially instead of in parallel.

#602 **FORALL executed serially (parallel array reference in FORALL)**

#603 **FORALL executed serially (triplet not permitted with mask)**

#604 **FORALL executed serially (functions of FORALL indices not allowed)**
Equations of FORALL indices are not allowed as subscripts; for example:
`FORALL (i=1:n) a(i) = b(2*i+1)`

#605 **FORALL executed serially (FORALL index repeated)**
For example: `FORALL (i=1:n) a(i,i) = i`

#606 **FORALL executed serially (FORALL index not in header order)**
For example: `FORALL (i=1:n, j=1:m) a(j,i) = i*j`

#608 **FORALL executed serially (vector array reference without FORALL indices)**

#609 **FORALL executed serially (array reference does not contain all FORALL indices)**
For example: `FORALL (i=1:n, j=1:m) a(i,j) = B(j)`

#610 **FORALL executed serially (mask format unsupported)**

#611 **FORALL executed serially (transformational intrinsic in FORALL)**

#612 **FORALL executed serially (front-end array reference with FORALL subscript(s))**

#613 **FORALL executed serially (external function)**

#614 **FORALL executed serially (array constructor with FORALL subscript(s))**

#615 **FORALL executed serially (triplet subscript with FORALL subscript(s))**

#616 **FORALL executed serially (statement function with FORALL subscript(s))**

#617 **FORALL executed serially (intrinsic function with FORALL subscript(s))**

#619 **FORALL executed serially (expression too complex)**

#620 **FORALL executed serially (triplet before FORALL index)**

Note that a function cannot be called in a parallel FORALL statement. Try inlining the function for better performance. See Chapter 4 of the *MasPar Fortran Reference Manual* for more information on FORALL.

Scalar Array I / O

This occurs when a read of an array cannot use parallel I / O. This occurs with noncanonical mapping, arrays that are too small on the DPU, and some array subscripts. Review the program and modify those sections that are causing it to run serially.

#801 **array I/O done in scalar mode**

Review your program to understand why the I / O is happening in serial instead of parallel fashion. Refer to the discussion of parallel I / O on page 4-19, and to the attachment to the System Software Release Notes called "Maximizing I / O Performance". Also, see Chapter 2 for more information on array mapping.

Strip Mining

These messages are relevant if you use the –strip=schedule command-line option. They indicate where the option takes effect, or where/why not.

#901 **strip loop not load scheduled, too many PE registers req'd**

#902 **load scheduling strip loop**
This message will appear if you use the –threshold=1 or higher option.

Loop Fusion

This message is relevant if you use the –Omax or –strip=fusion options. (It will also appear if you use the –threshold=2 option.) It indicates where loop fusion optimization is taking place.

#1001 **loops fused**

Blocked Algorithm Inhibition

These messages are relevant if you use the –blocked option. (They will also appear if you use the –threshold=2 option.) They assist in identifying where a program should be operating on a machine-sized array, but actually is not.

#1251 **strip-mined 2 level(s) deep**

#1253 **array '<name ' has <n layers**

Storage

These messages are relevant if you use the –storage option. These diagnostics help in determining memory usage on the DPU, and indicate where temporary memory was allocated by the compiler. (These messages also appear with the –threshold=1 or higher option.)

#1801 **array temp made on DPU for PE array mapping alignment**

#1802 **array temp made on DPU to handle loop dependence**

#1803 **array temp made on DPU**

#1804 **array temp made on DPU for I/O**

#1805 **array temp made on DPU for array section actual argument**

#1806 **array temp made on DPU for sloshing array for foreign call**

#1807 **array temp made on DPU for sloshing a dummy argument**

#1808 **array temp made on DPU for an automatic array**

#1809 **array temp made on DPU for serializing a FORALL statement**

#1810 **array temp made on DPU for a transformational intrinsic**

#1811 **array temp made on DPU for vector-valued subscript**

#1812 **array temp made on DPU for an array mapping hotel**

#1813 **array temp made on DPU for an array constructor**

#1814 **array temp made on DPU for a sequential array constructor**

#1815 **array temp made on DPU for an array inquiry intrinsic**

#1816 **array temp made on DPU for a return value**

#1817 **array temp made on DPU for an array expression actual argument**

#1818 **array temp made on DPU for a WHERE statement**

#1819 **array temp made on DPU for a reduction intrinsic**

#1820 **array temp made on DPU for a shift intrinsic**

#1821 **array temp made on DPU for a SPREAD intrinsic**

#1822 **array temp made on DPU for a MATMUL intrinsic**

#1823 **array temp made on DPU for a MERGE intrinsic**

#1824 array temp made on DPU for a RESHAPE intrinsic

#1825 array temp made on DPU for a PACK intrinsic

#1826 array temp made on DPU for an UNPACK intrinsic

#1827 array temp made on DPU for a MAXLOC/MINLOC intrinsic

#1828 array temp made on DPU for a TRANSPOSE intrinsic

#1829 array temp made on DPU for an intrinsic function

Execution Profiling

Execution profiling tells you how much time your program *actually* spends executing each routine and statement, based on the number of ticks each routine and statement accrues. Your program gets two ticks (one for the front end and one for the ACU) each time it receives a UNIX clock interrupt. Using execution profiling, you can look for places where you can improve your program's execution speed through optimization of your code. There are two types of execution profiling: flat and hierarchical.

With a **flat profile**, the two ticks are credited only to the routine and statement executing at the time of the clock interrupt. Routines compiled with flat profiling do *not* include the time spent in calls to other routines. With a **hierarchical profile**, the two ticks are credited not only to the executing statement and routine, but also to the lowest level hierarchically profiled routine that called the executing routine. (See the *Commands Reference Manual* for details on these types of profiles.)

The type of profile you get for a routine depends on how it has been compiled. You get a hierarchical profile if the routine has been compiled for hierarchical profiling (the –hprofile option). With the –nohprofile option you get a flat profile. The MPF compiler provides hierarchical profiling by default.

Note: There could be a minor performance impact when compiling with the default. This impact is usually quite negligible; however, to avoid it, compile with the -nohprofile option.

Hierarchical profiling is useful for finding where your program is incurring overhead for both library routines and your own routines. Use hierarchical profiling to determine the following:

- if and where your program spends time copying arrays between the ACU and front end
- how much time your program spends in an intrinsic routine
- how much time your program spends in explicit calls to library routines, like MPML and MPDDL

You can use hierarchical profiling effectively at various levels in your program. You can start by compiling all of your code with hierarchical profiling. Then, as you fine-tune

your lowest level routines, turn off hierarchical profiling on those routines. This results in the time for those low-level routines being added to the profiles for the remainder of your routines. Continue turning off the profiles at various levels until you are at your main routine. This approach allows you to make maximum use of hierarchical profiling throughout the tuning phase of your development.

For more information on mpprof, see the *Commands Reference Manual*. For more information on MPPE, see the *MPPE User Guide*.

Runtime Error Checking of Array Mapping

The DPU does not provide the linear memory environment found on the front end. Instead, it provides a processor grid. As a result, a program like the one below is illegal, because an array that is used on the DPU will be remapped across the subroutine interface from a 10x10 rank two array to a 100-element rank one array.

```
program illegal
real, dimension(10, 10) :: a

     a = 0.0              ! "a" is on the DPU
     call bad_idea( a )
end

subroutine bad_idea( a )
real, dimension( 100 ) :: a

     a = a + 1
end
```

Remapping arrays that are used *only* in Fortran 77 statements within a single routine *is* legal and will work properly.

MasPar Fortran inserts two kinds of runtime checks to detect cases where you have inadvertently remapped a DPU-resident array across a subroutine boundary. The first check is called **rank check**. This check is *always* done and ensures that the number of dimensions in the argument matches the number of dimensions in the local declaration.

The second check is referred to as **extent check**. This check is *not* done when the –Omax option is used. Extent check ensures that the dimensions of an array argument match the dimensions in the local declaration. The rank and extent runtime checks are *not* done for arrays that are used only in Fortran 77 statements within a single routine.

When one of these checks finds a mismatch, a runtime error will result. You should then examine the code, using MPPE to find which routine caused the error.

Timing Program Performance

Two timing routines, mpTimerStart() and mpTimerElapsed(), assist in analysis of MasPar Fortran performance. mpTimerStart() initializes the clock. mpTimerElapsed() is an integer function that returns the number of milliseconds of elapsed time since the last call to mpTimerStart(). To use these routines effectively, simply bracket the code to be timed with the routines. For example:

```
integer foo

call mpTimerStart()       ! starts the clock going
do i = 1, 10000000
        call sub(i)
end do
foo = mpTimerElapsed()    ! number of milliseconds since
print *,'foo=', foo       ! mpTimerStart
end

subroutine sub(k)
integer k
end
```

These routines use the UNIX system clock through gettimeofday(2), so are accurate to a few msecs. Two similar MPL routines, dpuTimerStart() and dpuTimerElapsed(), use the MasPar system clock (80 nsecs), and so can provide a finer level of analysis. In addition, they more accurately measure runtime for a job that is sharing the DPU. A limitation of these routines is that they can only measure runtimes of less that ~320 secs before the internal counter overflows.

Because MasPar Fortran can call MPL, a MasPar Fortran program can get at these routines by adding jacket routines. These jacket routines would look like the following:

```
#include <mpl.h>

extern double dpuTimerElapsed();

DPUTIMERSTART()
{
        dpuTimerStart();
}

DPUTIMERELAPSED(feArgs,dpuArgs)
        long *feArgs[];
        void *dpuArgs[];
{
        double *et_ptr;
        double et;

        et = dpuTimerElapsed();
        copyIn(&feArgs[1], &et_ptr, sizeof(et_ptr));
        copyOut(&et, et_ptr, sizeof(et));
}
```

Now you can call these routines from MasPar Fortran:

```
            program example
            real*8 et
            real a(256,256), b(256,256)
CMPF        MPL dpuTimerStart
CMPF        MPL dpuTimerElapsed

            call dpuTimerStart()
            a = 1
            b = 2
            a = a * b
            call dpuTimerElapsed(et)
            print *, et
            end
```

Table Lookup with FORALL

The MPF compiler supports the table-lookup feature (also available with MPL) by way of the FORALL statement.

"Table-lookup" means you can do processor-local addressed array indexing, as in:

```
plural int   B;
plural float TAB[10];
plural float A;

A = TAB[B];
```

In each processor a different value of B is being used to index the array TAB, and the array TAB can have different values on each processor.

A true semantic equivalent in MasPar Fortran uses vector-valued subscripts, as in:

```
integer   B(1024)
dimension TAB(10,1024)
dimension A(1024)

forall (i=1:1024) A(i) = TAB(B(i),i)
```

In addition, if the array TAB is mapped:

```
cmpf map TAB(memory,allbits)
```

then the compiler will not generate Global Router code; instead, it generates the same code as the MPL compiler for processor-local-addressed array indexing.

The example can be extended to two dimensions, as in:

```
forall ( i=1:32, j=1:32 ) A(i,j) = TAB( B(i,j), i,j )
```

In this case, to get good performance (no router use) the array TAB should be mapped as:

```
cmpf map TAB(memory,xbits,ybits)
```

In general, to avoid routers in these FORALL vector-valued subscript statements, map the table array being indexed to memory in the dimension that has the vector-valued subscript; map all other dimensions identically to the way the vector-valued subscript is mapped.

If you are in doubt about a particular instance, compile it with the –report command-line option and look for the message:

```
"line 42: Table Lookup (vvss without router!)."
```

Notice that the table array's size can be large. In some applications the values of the table will be the same across all processors; it is important to not waste space, but also important to not use the router. In this case a blocked algorithm will work:

```
        dimension TAB(10,NPROC)      !! machine sized lookup table
cmpf map TAB(memory,allbits)         !! mapped for no router table lookup
        dimension ATEMP(NPROC)       !! machine sized temp
        integer   BTEMP(NPROC)       !! machine sized temp

        do i=1,NMAX,NPROC
          BTEMP = B(i:i+NPROC-1)
          forall (ii=1:NPROC) ATEMP(ii) = TAB(BTEMP(ii),ii)
          A(i:i+NPROC-1) = ATEMP
        enddo
```

By sizing the problem to the machine size, better performance can be achieved. This is due to the fact that the MasPar Fortran compiler now maps arrays that are less than or equal to the machine size into PE registers instead of into PE memory. If the array is larger than the machine size, the whole array is stored in PE memory. The compiler decides how to store the array data based on the size of the array. If the compiler does not know the size of the array, it will store it in PE memory. To take advantage of this, write your code so that the problem matches the machine size. Using a blocked algorithm, like that shown above, can help you do this.

Getting High Performance I / O

The I / O throughput provided by the MasPar parallel disk array (MPDA) is much higher than the throughput to the disk attached to the front end. MasPar Fortran programs can take advantage of this higher throughput by using sequential unformatted reads and writes. Formatted reads and writes currently use sequential I / O, even when the file is opened on the disk array.

Files on the disk array are standard UNIX files and can be accessed by both the front end and the DPU. However, high performance I / O is only available when the array is in PE memory on the DPU. Because there is significant overhead in moving arrays between the front end and the DPU, there is little advantage in moving data from the front end for parallel I / O. Therefore, MasPar Fortran uses front-end I / O routines when the array is on the front end. When an array is allocated on the DPU, parallel I / O routines are used; thus I / O throughput is much higher. Within a given routine, arrays are allocated on the DPU when they are used in array expressions or when they are explicitly placed on the

DPU with an ONDPU compiler directive. An ambiguous case exists for subroutines and functions where the only reference to an argument array is in an unformatted I / O statement. Whether or not parallel I / O will be used in this case depends on if the array was on the DPU in the calling routine.

The subroutine shown here opens a file on the disk array, writes an array to the file, then reads the array back in. After the read, it verifies that the expected data was returned.

```
  subroutine whole_array_IO
  implicit none
  integer, parameter :: dim1 = 1000, dim2 = 1000, dim3 = 3
  integer, array(dim1, dim2, dim3) :: a
  integer i, iorslt
  logical passed
  character*10 file_stat
  character*60 path

    passed = .true.
    ! the array "a" is used in an array expression, so it
    ! will be allocated on the DPU
    do i = 1, dim3
       a(:, :, i ) = i
    end do
    path = "/da/da0c/whole_array"
    file_stat = 'DELETE'
    ! open file for unformatted I/O
    open(unit = 10,
&        form = 'UNFORMATTED',
&        iostat=iorslt,
&        file = path )
    if (iorslt <> 0) then
       print *, 'whole_array_IO: error opening file'
       print *, 'iorslt = ', iorslt
    end if
    write(10) a
    rewind(10)  ! go back to beginning of file
    read(10) a
    verify: do i = 1, dim3
       if (any(a(:, :, i ) <> i)) then
          print *, 'whole_array_IO: bad data'
          passed = .false.
          exit verify
       end if
    end do verify
    if (passed) then
       print *, 'test passed'
    else
       print *, 'test failed'
    end if
    close(10, status = file_stat ) ! file should be removed by the close
  end
```

For more information on parallel I/O, see the attachment to the System Release Notes called "Maximizing I/O Performance".

Chapter 5

Converting Fortran 77 Programs

Subject to certain constraints, existing Fortran 77 programs will run on the front end of the MasPar system. However, to take advantage of the data-parallel processing capabilities of the MasPar system you will need to change some Fortran 77 constructs to MasPar Fortran constructs. This section discusses how to modify your existing Fortran 77 programs so they will access the DPU effectively.

Not all parts of a Fortran 77 program will necessarily need to be modified; only those parts of the program that would benefit from parallel execution should be analyzed and recoded. Before beginning to convert your program to MasPar Fortran, you should examine your entire algorithm to identify all calculations that would benefit from parallel processing, as well as areas that could benefit from restructuring or reorganization. Merely modifying isolated constructs without first reviewing your entire algorithm might not give you the most efficient performance possible.

Before you begin to optimize your code, you must make sure it will run properly on the MasPar system. Following are some guidelines for successful conversion of your Fortran 77 programs to MasPar Fortran. Note that some of these suggestions might not apply to your program, depending on which implementation of Fortran 77 you are converting.

Using Conversion Tools

Converting existing Fortran 77 applications to MasPar Fortran can be time-consuming. This process can be eased and speeded by use of automated conversion tools. MasPar VAST-2, a sophisticated Fortran 77-to-Fortran 90 source code translator, is an invaluable aid in porting Fortran applications to the MasPar system. With MasPar VAST-2 you can eliminate much of the tedious work of code conversion.

MasPar VAST-2 is an optional product. Contact your MasPar Sales representative for more information on MasPar VAST-2.

Converting Your Program Manually

If you do not have a program conversion tool, you will need to convert your program manually. These sections describe the basic steps to successful conversion.

Preparing to Run Your Program on the Front End

Before changing any of your Fortran 77 code to MasPar Fortran code constructs, make sure your program compiles and runs successfully on the front end of the MasPar system. Using the compiler for which your program was originally written, make the following changes, compiling and executing regularly to ensure the changes are not creating new errors.

- If your source file contains multiple subprograms, split the individual program units into single files. This will reduce compile time, because you will not have to recompile units that are working properly. Also, it will make it easier to localize problems.
- Look through your program to find any nonstandard Fortran 77 constructs—these are implementation-specific extensions to Fortran 77 that are peculiar to the machine for which your program was originally written. These include things such as Hollerith constants, the use of byte and integer*2 types, and so forth. Change these to standard Fortran 77 constructs.
- Check your program for any local variables that must maintain their values between calls. These variables must be declared as SAVE variables. MasPar Fortran (like Fortran 77) does not provide the SAVE attribute by default. If your original Fortran compiler has an option to make local variables automatic, compile with that option *on* to ensure that you have marked all the appropriate variables.
- Check the *MasPar Fortran Release Notes* to see if there are any other changes you should make at this point.

Your program should compile and execute successfully on the front end with your original compiler (e.g., f77).

Preparing Your Program to Handle COMMON Arrays

If your program does not have any arrays in COMMON blocks, skip this section.

By default, the MasPar Fortran compiler allocates *all* COMMON arrays on the front end, regardless of how they are used. This allows Fortran 77 COMMON blocks to compile successfully without modification. However, this default will cause the compiler to flag as an error any COMMON arrays used in Fortran 90 expressions, because the COMMON array remains where it was allocated (by default on the front end), regardless of how it is used. Therefore, the Fortran 90 array will not be moved to the PE array to compute an array expression.

When you use COMMON arrays in Fortran 90 array expressions, you must use the –nofecommon option when compiling the routines in your program. This option forces all COMMON arrays to be allocated on the PE array, allowing them to be used in Fortran 90 expressions. To use the –nofecommon option, be sure your COMMON blocks conform to these guidelines:

- Remove all EQUIVALENCE statements that refer to arrays in COMMON. *You cannot assume a linear memory structure on the PE array like you can on the front end.*
- Make all COMMON blocks consistent in each routine that uses them. All arrays in a given COMMON block must be declared with the same size and extents in each subprogram where the COMMON block appears. This is true for both blank and named COMMON blocks. One way to ensure consistency is to use the INCLUDE feature to include a single COMMON block definition in each subprogram.
- Split up existing COMMON blocks so that they do not have mixed classifications. Each COMMON block can have only one of three kinds of variables:
    1. numeric or logical arrays
    2. numeric or logical scalars (that is, non-arrays)
    3. characters
- Use large arrays only in array expressions or move them out of COMMON. If a large array is used in a scalar expression when the –nofecommon option is used, the compiler must move each element of the array in the scalar computation from the PE array to the front end. This overhead can significantly slow down execution.

Note that MasPar VAST-2 can automatically remove EQUIVALENCE, check COMMON blocks, and split out scalars and arrays into separate include files.

Now compile your program with mpfortran and the –nofecommon option. It should run successfully, with the compiler allocating array data held in COMMON to the DPU.

Use MPPE to profile and examine your program.

Converting Your Program to Use Fortran 90 Array Features

You are ready to begin modifying your code to use the parallel capabilities of the DPU to improve performance and efficiency. Use MPPE to determine which portions of your program are creating performance bottlenecks, and concentrate on modifying them first.

- Modify your Fortran 77 array notation to Fortran 90 syntax where arrays are passed that are not shape-conforming. Change the way slices of arrays are passed in argument lists.
- Use the array sectioning and assumed-shape array features when passing arrays as arguments.
- Use the MasPar Fortran intrinsics instead of user-defined functions wherever possible.
- Use the ONDPU and ONFE directives and the mapping directives to effectively control data placement.

Use the compile-time and execution profiling tools available with mpfortran, mpprof, and MPPE to tune your program (see Chapter 4).

See Chapters 2 through 4 for more details on developing MasPar Fortran programs. Also, see the *MasPar Fortran Reference Manual.*

Chapter 6

Compiling MasPar Fortran Programs

This chapter introduces the MPF compiler and describes the command syntax and command-line options you use to invoke it. Command-line information can also be accessed online by typing

```
man mpfortran
```

at the system prompt.

Functions of the Compiler

The MPF compiler, the assembler, and the linker are invoked and controlled by a single command: mpfortran. If there are no compilation errors, mpfortran invokes the assembler and the linker. The object files created by the assembler, and any linker options or object files specified on the command line, pass to the linker.

The main functions of the MPF compiler are as follows:

- Verifying that the MasPar Fortran source statements are correct, issuing error messages if necessary.
- Determining which statements and constructs in the source program should be processed on the front end and which should be allocated on the DPU.
- Generating code to move data from the front end to the DPU, and vice versa, for most efficient processing.
- Generating machine language instructions from the source statements in the program.
- Grouping instructions into an object module that can be processed by the linker.

The mpfortran Command

The mpfortran command is the user interface to the MPF compiler. It accepts a number of options and filenames and invokes one or more programs to operate on the file (the compiler, the MasPar assembler, and the linker).

You specify the name of your MasPar Fortran source file on the mpfortran command line. The file is then processed by the MPF compiler, and the resulting object file is passed to the assembler and the linker, along with appropriate runtime library routines, to produce an executable object file. However, you can select from many processing options and specify files other than MasPar Fortran source files. The combination of the processing options and the type of file specified determines how the mpfortran command handles the processing.

Syntax

To invoke the MPF compiler, use the command mpfortran. The command-line syntax is

mpfortran [*–options* ...] *files* ... [*–options* ...]

Specifying Input Files

You must give the full file specification on the mpfortran command line; that is, if the files that you specify are not in your working directory, you must explicitly specify their directory location.

If you specify multiple files, use spaces to separate the filenames. Commas or other special characters are interpreted as part of the filename.

The mpfortran command accepts the following types of arguments as filenames:

- Arguments whose names end with .f, .F, .for, or .FOR are taken to be MasPar Fortran source programs. The compiler compiles the source code, and the assembler puts the resulting object code in a file whose name is the same as the name of the source file, except .o is substituted for .f, .for, or .FOR.
- Arguments whose names end with .S are taken to be MasPar assembly source programs and are passed to the MasPar assembler, producing a .o file.
- Arguments whose names end with .o or .a are taken to be object files and are passed to the linker.

Arguments other than those ending with .f, .F, .for, .FOR, .S, .o, or .a are treated as one of the following:

- compiler or linker option arguments
- object programs that were produced during an earlier compilation or that were extracted from the libraries

If you specify only one source file without specifying the –c option, the compiler does not create a .o file. If you have multiple source files, the compiler will always create a .o file for each source file.

All assembly and object files produced by the mpfortran command, by default, contain debug information and compile-time analysis messages for subsequent use by MPPE and mpprof(1). This information can be suppressed by using the –nodebug option. For a discussion of the compile-time analysis messages, see Chapter 4.

Specifying Options on the Command Line

Options to the mpfortran command affect the way the compiler processes the source file. Options are usually necessary only when special processing is needed.

You specify options on the command line with an option string preceded by a dash (similar to the option style for the UNIX operating system).

The following options are available with the current release of MasPar Fortran:

–[no]blocked	Force the –report option, printing only compile-time analysis messages that indicate multilayer array mappings

	that might affect blocked algorithms. Ignore the –threshold option, and print messages of all levels. The default is –noblocked.
–c	Suppress linking and produce .o files for each source file.
–continuations=*n*	Where *n* is an integer from 0 to 99 specifying the number of continuation lines allowed in a source program statement. The default allowed is 19 continuation lines.
–[no]cross_reference	If the –V option is specified, include the numbers of the lines in which symbols are defined and referenced in the storage map section of the listing file. The default is –nocross_reference.
–nodebug	Do not generate debugging information. This option is useful when you have "out of memory" problems. You cannot use MPPE or mpprof pm code compiled with this option, however.
–[no]d_lines	Compile lines with a `d` or `D` in column 1; do not treat them as comment lines. The default is –nod_lines.
–double_precision	Use double precision as the default precision for REAL and COMPLEX variables. Declarations of REAL and COMPLEX types are compiled as DOUBLE PRECISION and DOUBLE COMPLEX. (REAL*4 and COMPLEX*8 declarations are not affected by this option.)
–[no]extend_source	Extend the range of statement text from columns 1–72 to columns 1–132. The default is –noextend_source.
–[no]fecommon	Make all COMMON data resident on the front end for COMMON blocks that can contain both scalars and arrays (array operations on these arrays are not allowed). Use this option during initial porting of Fortran 77 programs to MasPar Fortran to avoid having to immediately modify all COMMON blocks that mix scalars and arrays. The default is –fecommon; –nofecommon puts COMMON array data on the DPU. (See the section "Controlling COMMON Array Data Allocation on the Front End and DPU" on page 2-2.)
–[no]hprofile	Generate extra code to write hierarchical profile information for analysis under MPPE or with mpprof(1). The default is –hprofile. Use –nohprofile to suppress generation of this extra code. See the MPPE manual set and the *Commands Reference Manual* for details on hierarchical profiling.
–include=*path*	If an include file is not found in the current working directory, search *path* for source include files, in the order specified.
–L*dir*	Add *dir* to the list of directories that are searched for libraries. Directories specified with –L are searched before the standard directories.

–l*string*	Search for a library named *string*, which is a file named lib*string*.a, where *string* is a string. The MasPar linker searches for libraries first in any directories specified with –L options, then in the standard directories. The first default library searched is $MP_PATH/lib/maspar, if MP_PATH is set, or /usr/maspar/lib/maspar, if MP_PATH is not set. The remaining default libraries searched are /lib, /usr/lib, and /usr/local/lib. A library is searched when its name is encountered, so the placement of a –l in the compiler or linker command line is significant. The math (–lm) and C (–lc) libraries are automatically linked in; it is not necessary to specify them on the command line.
–o *output*	Name the final output file *output* instead of *a.out*.
–O	Compile with optimization (the default). Some debugger features are not supported at this level of optimization (for example, assigning values to variables while debugging). In this release of MasPar Fortran, the –O option yields the same result as –Omin.
–Omax	Compile with maximum optimization. This option provides the most optimization available, but none of the debugger features are supported.
–Omin	Compile with minimum optimization. This option allows the use of all debugger features. In this release of MasPar Fortran, the –Omin option yields the same result as –O.
–pesize=*integer*	Specify the size of the DPU PE array for which code will be compiled; the default is 1K PEs. Valid integer values are 1, 2, 4, 8, 16, meaning 1K, 2K, 4K, 8K, 16K PEs. See the section "Specifying the Target PE Array Size" on page 6-6 for more information.
–pevariable	Compile code for any size machine. Arrays in COMMON and arrays with the SAVE attribute *cannot* be used with this option. By default, the –pesize option is in effect.
–pmemsize=*integer*	Specify the size of the PE memory for error checking. By default, the amount of memory assumed in each PE is 64 KBytes. Any value of <*integer*> greater than 16,384 is treated as 64 KBytes; any value up to 16,384 KBytes is treated as 16 KBytes. The compiler verifies that the known amount of PE storage used by any routine does not exceed the size specified; it generates an error message if it does.
–[no]report	Print a report of the compile-time analysis messages of level 0 for each subprogram on standard output. (See the –threshold option for more information on levels.) If the –V option is specified, include the messages in the listing file also. The default is –noreport.
–S	Compile programs and write output to .S files.

–save_all_variables	Retain values of all local arrays and variables after procedure return (as if specified in the SAVE statement). By default, local arrays and variables are not saved. This option is useful for debugging. Note that using this option has high costs at runtime and in DPU memory usage, and normally should not be used.
–save_fe_variables	The same effect as –save_all_variables, except it does not apply to the DPU (it retains front-end values only). As with –save_all_variables, this option is useful for debugging, but normally should not be used, due to high runtime and memory costs.
–[no]storage	Force the –report option, printing only compile-time analysis messages that indicate storage usage. Ignore the –threshold setting and print messages of all levels. The default is –nostorage.
–strip=schedule	Perform load/store scheduling optimizations. To have an effect, this option must be used in conjunction with the –Omax option, which must be separately specified. The compiler does overlapped arithmetic and memory operations in strip loops (compiler-generated loops when the arrays being operated on are larger than the machine size).
–threshold=*n*	Force the –report option, printing only compile-time analysis messages with a level less than or equal to *n*. The default is *n*=0. As a general rule, as the level number increases, the messages become less critical and might be more voluminous. Currently, the highest level used is 2.
–[no]v104	Use intrinsic arguments and syntax from the V1.0 MPF compiler, which was based on V104 of the draft Fortran 90 standard. The current compiler has updated syntax and arguments of intrinsics to comply with the approved ISO standard. The default is –nov104.
–Zq	Suppress the MasPar copyright and version number notice.

Specifying the Target PE Array Size

Use the –pesize option to specify the number of parallel processors in the DPU for which you are compiling your code. Depending on the type of MasPar system you are using, "pesize" can equal 1, 2, 4, 8, or 16. For example, –pesize=1 means that there are 1,024 parallel processors in your target DPU.

When –pesize=	*there are*	*and the machine size is*
1	1,024 processors	32 columns x 32 rows
2	2,048 processors	64 columns x 32 rows
4	4,096 processors	64 columns x 64 rows
8	8,192 processors	128 columns x 64 rows
16	16,384 processors	128 columns x 128 rows

NOTE: The default is 1K PEs; therefore, if your machine has a different capacity, you should use this option.

Alternatively, using the –pevariable command-line option lets you compile a MasPar Fortran program into a single binary executable that will run on all MasPar machine configurations.

Compiling with Optimization

The compiler supports three levels of optimization with command-line options –Omin, –O, and –Omax. By default, the MPF compiler performs all –O optimizations. These include optimizations like common subexpression elimination, value propagation, and global register allocation. The code generated by the MPF compiler can be debugged with MPPE when these optimizations are present. The –Omax option compiles the program with all the optimizations currently available and allows profiling in MPPE.

Floating Point Faults

The current version of the MPF compiler causes a trap by default on

- invalid operands
- overflows
- divide-by-zero

It does not trap on underflow conditions or inexact results (note that the front-end F77 compiler *does* trap on underflow). This default differs from earlier releases of the compiler, which did not trap on divide-by-zero cases.

On the DECstation, you can control the trap conditions by using the ULTRIX library function set_fpc_csr() (see the online man page). Also, calling the function mpfNoTraps() will turn off all traps, both on the DPU and on the front end.

Chapter 7

Calling Other Languages

From MasPar Fortran you can call routines written in DEC Fortran (f77), front-end C cc, and MPL (MasPar Programming Language).

You can take advantage of previously existing user routines and standard library routines by calling DEC Fortran 77 or front-end C.

When execution speed is critical, you can improve the performance of a MasPar Fortran program by coding the most computationally intensive routines in MPL, and then calling these routines from MasPar Fortran. MPL, based on ANSI C, is the lowest level programming language that MasPar supports. MPL gives the user direct control over the machine.

This chapter assumes that you are experienced in programming with the language you want to call (MPL, DEC Fortran, or front-end C). For details on MPL, see the *MasPar Programming Language (MPL) User Guide* and *Reference Manual.* For details on DEC Fortran and front-end C, see the documentation provided by DEC.

Calling Front-end Routines from MasPar Fortran

With some restrictions, you can call user-written front-end Fortran 77 and C routines and the DEC MIPS Fortran libraries from within your MasPar Fortran programs.

User-Written Front-end Routines

When converting existing programs to MasPar Fortran, the most computationally intensive routines should be converted to Fortran 90-style array constructs. Routines that do not involve heavy computation, or that use features not available in MasPar Fortran, can be left in the original front-end language and called from within your MasPar Fortran program. This eases program conversion.

By identifying them in your MasPar Fortran program with a compiler directive, you can call DEC Fortran (f77) and front-end C (cc) routines from MasPar Fortran.

Identifying Front-end Routines to the MPF Compiler

Front-end language routines must be identified with a compiler directive in the routine (or main program) in which the front-end routine is invoked. The syntax for this directive is

[*comment_directive*] [*language*] *routine_name* [, *routine_name*, ...]

where

comment_directive	Is cmpf. It must start in the first character column, and there can be *no* spaces between the characters. Any other comment (for example, c mpf) is treated as a comment and is ignored by the MPF compiler.
language	Follows the comment directive and is the name of the language of the routine being called. It is case-insensitive. The directive F77 or f77 identifies DEC MIPS Fortran 77 routines. The directive C or c identifies front-end C routines.
routine_list	Consists of the names of the routines being called, separated by commas.

Although the directive can be used with routines declared in Interface blocks, the directive itself *cannot* be included inside the interface declaration. Several routines can be listed in one directive statement, but the directive *cannot* be continued over more than one line. Following are some examples:

Example:

```
      interface
        subroutine foo( a, b )
        integer a, b
        end

        real function bar( x )
        real x
        end
      end interface

cmpf  F77 foo, bar          ! two front-end Fortran routines

        call foo( p, q )
        z = bar( y )
```

Example:

```
        character*10 num
        character*10 int_to_char
        external int_to_char

cmpf    F77 int_to_char     ! one front-end Fortran function

        num = int_to_char( i )
```

Some Example Calls

In the following example, an array that is resident on the DPU (because it is used in a Fortran 90 expression) is passed to the front-end DEC Fortran 77 routine dpu_sliced_array.

```
      integer, parameter :: dim1 = 100, dim2 = 50
      integer, dimension( dim1, dim2 ) :: a
cmpf  F77 dpu_sliced_array  ! calls 1 front-end Fortran routine

        a = 0
        call dpu_sliced_array( a(1:dim1:2, 1:dim2:2),
     &                         dim1/2, dim2/2 )
```

Before the call, the array argument is evaluated and its values moved to the front end. The address of these values in front-end memory then is used as the argument in the call. After the call, the front-end values (which might have been changed by the front-end code) are moved to the DPU.

Be aware that the overhead in moving data between the front end and the DPU can be significant.

You can reduce this overhead by declaring the front-end subroutine or function in an Interface block and defining the array argument with the attribute INTENT(IN) if it is only referenced in the routine, or INTENT(OUT) if the array argument is only written to. For example:

```
      interface
        subroutine relax( a )
        real, dimension( 100 ) :: a
        intent( in ) :: a
        end

        integer function tense( b )
        integer, dimension( 20 ) :: b
        intent( out ) :: b
        end
      end interface
cmpf    F77  relax          ! calls 1 front-end Fortran routine
cmpf    C    tense          ! calls 1 front-end C routine
cmpf    ONDPU  a, b         ! forces arrays a and b on the DPU
```

The arrays a and b are allocated on the DPU. When subroutine relax is called, the values in a are moved from the DPU to the front end, but the values on the front end will not be moved back to the DPU after the call. When function tense is called, the value on the DPU is *not* moved to the front end. However, after the call the values on the front end are copied to the DPU.

As long as the common block is on the front end, both named and blank common can be referenced from either MasPar Fortran or front-end Fortran 77. If the common block is on the DPU, a link error will result.

In the next example, the array a is allocated on the DPU, because it is used in a Fortran 90 array assignment. When the array element a(i) is referenced, only that element is moved back and forth between the front end and the DPU, not the entire array. If subroutine foo attempts to treat its dummy argument as an array address instead of as a scalar, the program will not yield a correct result.

```
      integer, parameter :: dim1 = 100, dim2 = 50
      integer, dimension( dim1, dim2 ) :: a

cmpf  F77 foo

        a = 42
        call foo( a(i) )
```

Input / Output

MasPar Fortran and DEC Fortran 77 use different runtime libraries to support input / output operations. If one side (front-end Fortran 77, for example) attempts to use a unit number opened by the other side (MasPar Fortran, for example) the I / O operation will fail. However, redirection of UNIX standard-in and standard-out will work for both sides.

Front-end C Considerations

Following Fortran convention, MasPar Fortran passes all values by reference (that is, by address). This is true even when the routine being called is identified as a front-end C routine with the C compiler directive. In the C function declaration the arguments must be declared as pointers. For example:

MasPar Fortran:

```
      integer i
      real r
      double precision d
      complex cx
cmpf  C bar

        call bar(i, r, d, cx)
```

Front-end C:

```
typedef struct { float real;
                 float imag;
               } single_complex;

bar( i, r, d, cx )
  int *i;
  float *r;
  double *d;
  single_complex *cx;
{
  /* ... */
}
```

Front-end C functions can return integer, real, and double precision results to MasPar Fortran. However, character and complex results *cannot* be returned from C.

As long as the common block is on the front end (the default for MasPar Fortran), common blocks can be referenced from front-end C. In the following example, the common block plebeian can be referenced by declaring a global variable named plebeian_ in C. The names of the variables in the common block *cannot* be referenced. However, as shown in the example, a structure corresponding to the common block *can* be declared in C.

MasPar Fortran:

```
      integer i, j
      real x, y
      complex cx1, cx2
      common /plebeian/ i, j, x, y, cx1, cx2
cmpf  C set_common

        call set_common()
```

Front-end C:

```
typedef struct { int i, j;
                 float x, y;
                 single_complex cx1, cx2;
               } common_struct;

common_struct plebeian_;

set_common()
{
  /* ... */
}
```

If you reference a Fortran blank common block, declare a global named _BLNK__.

Character Variables with C and Fortran

If you use C, be aware that Fortran character variables do *not* follow the C convention of being null-terminated strings. Initialized character variables are "filled" with space characters when the assignment does not totally fill the character variable. In the following example, the character variable "ch" will contain the string "12345", followed by five space characters.

```
      character*10 ch

        ch = '12345'
```

When calling a front-end Fortran or C routine, MasPar Fortran uses the DEC MIPS Fortran 77 convention for character variables. In the next example, the length of the character arguments follows the argument list as value arguments:

MasPar Fortran:

```
      character*10 ch1
      character*20 ch2
      integer i
      real x
cmpf  C zorch

        call zorch( ch1, ch2, i, x )
```

Front-end C:

```
zorch( ch1, ch2, i, x, len_ch1, len_ch2 )
  char *ch1, *ch2;
  int *i;
  float *x;
  int len_ch1, len_ch2;  /* Note: values, not pointers! */
```

Although character arguments can be passed to front-end C functions, C functions with a "char *" result *cannot* be called from MasPar Fortran because the convention for returning character function results differs.

Front-End Fortran Libraries

You also can call the DEC MIPS Fortran front-end libraries from within your MasPar Fortran program. These routines are described in section 3F of the DEC *ULTRIX Programmer's Manual.* See that manual for a complete list of routines and details on their syntax.

Each routine you call must be identified with the F77 compiler directive. The following example calls the UNIX `fdate` routine (see `fdate` in section 3F of the *UNIX Programmer's Manual*):

```
        program test
        character*40 date
cmpf    F77 fdate

          date = " "
          call fdate( date )
          print *, 'date = ', date
        end
```

Any f77 or cc routines called from MasPar Fortran cannot be debugged from MPPE.

The 3F library routines are contained in the libU77.a and libI77.a archives. When compiling a program that references these routines you must link in these libraries. To do this, use the –lU77 and –lI77 options on the mpfortran command line. For example:

```
mpfortran zorch.f -o zorch -lU77 -lI77
```

Calling MPL from MasPar Fortran

This section assumes you are experienced in programming with MPL on the MasPar machine. For details on MPL see the *MasPar Programming Language (MPL) User Guide* and the *MasPar Programming Language (MPL) Reference Manual.*

Identifying MPL Routines to the MPF Compiler

Because the code generated to call an MPL routine differs from that generated to call a MasPar Fortran routine, you must notify the compiler that a given routine is an MPL routine. The syntax for this directive is

[*comment_directive*] [*language*] *routine_name* [, *routine_name*, ...]

where

comment_directive	Is cmpf. It must start in the first character column, and there can be *no* spaces between the characters. Any other comment (for example, c mpf) is treated as a comment and is ignored by the MPF compiler.
language	Follows the comment directive and is the name of the language of the routine being called (in this case, MPL or mpl).
routine_list	Consists of the names of the MPL routines, separated by commas.

For example:

```
cmpf  mpl foo, bar
```

These routines (foo and bar) then are called like any Fortran routine.

The scope of the MPL directive is the same as that of Fortran subroutines; that is, a single directive cannot be used to declare all MPL subroutine routines within a file. The directive is only applicable for the subroutine in which it appears. In the example below, subroutine foo will be declared as an MPL subroutine. Because the cmpf directive is in the same scope as foo_caller, subroutine bar will *not* be declared as an MPL subroutine, and the compiler will issue a warning message ("Unable to find symbol "BAR" for attribute setting.").

```
!
! Subroutine BAR will not be declared
! as an MPL routine
!
cmpf    mpl foo, bar

        subroutine foo_caller

          call foo
        end

        subroutine bar_caller

          call bar
        end
```

This example is corrected in the following rewritten code:

```
        subroutine foo_caller
cmpf    mpl foo

          call foo
        end

        subroutine bar_caller
cmpf    mpl bar

          call bar
        end
```

Argument Blocks

MasPar Fortran passes arguments to routines in **argument blocks**. A front-end argument block is passed on the front-end stack; an ACU argument block is passed on the ACU stack. Argument data begins in the second 32-bit word of the argument block. In the front-end argument block, the first word is reserved for the **argument count**. In the ACU argument block, the first word is left unused. In both argument blocks, this first word is followed by one 32-bit word for each actual argument, in the same left-to-right order in which they appeared in the subroutine call. That word is typically the address of the actual argument or the address of its descriptor, also known as a **dope vector**. The contents of the argument blocks are slightly different for the front end and the ACU and for the various argument types.

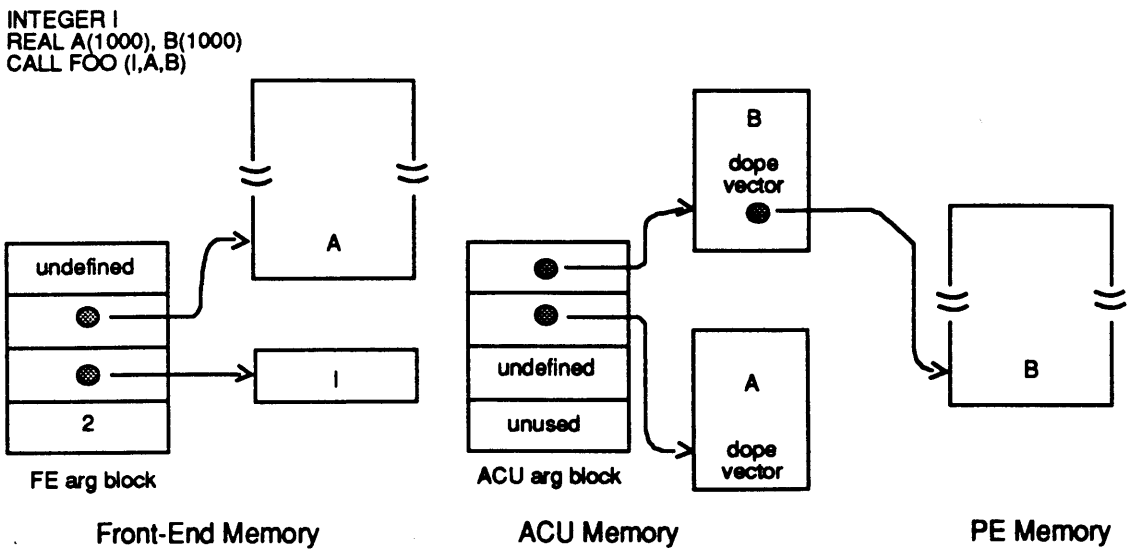


Figure 7-1 Front End and ACU Argument Blocks

For numeric scalar arguments ("I" in Figure 7-1), the address of the actual argument is in the front-end argument block, and the ACU argument block entry is undefined.

For a numeric array that has been allocated on the DPU ("B" in Figure 7-1), the address of the dope vector for the actual argument is in the ACU argument block, and the front-end argument block entry is undefined.

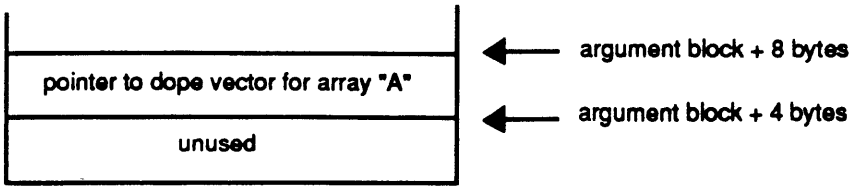
For a numeric array that has been allocated on front-end memory ("A" in Figure 7-1), the address of the array is in the front-end argument block, and the address of the dope vector is in the ACU argument block.

For example, when the MPL routine print_dope is called from the following code, an argument block for it is constructed in ACU memory.

MasPar Fortran:

```
      program mpltest
      integer, dimension( 10, 20 ) :: A

cmpf  mpl print_dope

        A = 42
        call print_dope( A )
      end
```

This argument block contains the dope vector that describes the array A.



The dope vector contains the address of the array, the rank of the array, the size of each of the array dimensions, and a flag indicating where the array is currently resident: on the front end or on the PE array.

Passing Data Between MasPar Fortran and MPL

The front-end and ACU argument blocks are the *only* arguments to an MPL routine, regardless of the number of arguments in the MasPar Fortran call. The arguments used in the MasPar Fortran call must be extracted from these argument blocks.

Following is an example in MPL. This code is called by the MasPar Fortran program mpltest shown in the preceding example. (Note that all MasPar Fortran routine names are converted to uppercase, even if they were originally in lowercase in the Fortran program.)

MPL:

```
typedef struct { plural int *addr;          /* array address */
                 unsigned int rank;         /* rank of the array */
                 unsigned int OnDpu;        /* TRUE if resident on PE array */
                 unsigned int extent[ 7 ]; /* sizes of the array dimensions */
               } DopeVector;

PRINT_DOPE( fe_arg_blk, dpu_arg_blk )
      void *fe_arg_blk[];
      DopeVector   *dpu_arg_blk[];
{
  DopeVector *DopePtr;
  int i;

  DopePtr = dpu_arg_blk[1];
  printf("rank = %d\n", DopePtr->rank );
  if (DopePtr->OnDpu)
    printf("OnDpu is TRUE\n");
  else
    printf("OnDpu is FALSE\n");
  printf("extents = [");
  for (i = 0; i < DopePtr->rank; i++)
    printf("%d ", DopePtr->extent[ i ] );
  printf("]\n");
}
```

To link the MPL routine with the MasPar Fortran program in this example, first compile the MPL and MasPar Fortran routines separately:

```
mpl -c print_dope.m
mpfortran -c -pesize=4 mpltest.f
```

and then link them using the mpfortran command:

```
mpfortran mpltest.o print_dope.o -o mpltest
```

After the MasPar Fortran program mpltest is compiled and linked with print_dope.m and then executed, it prints the following:

```
rank = 2
OnDpu is TRUE
extents = [10 20 ]
```

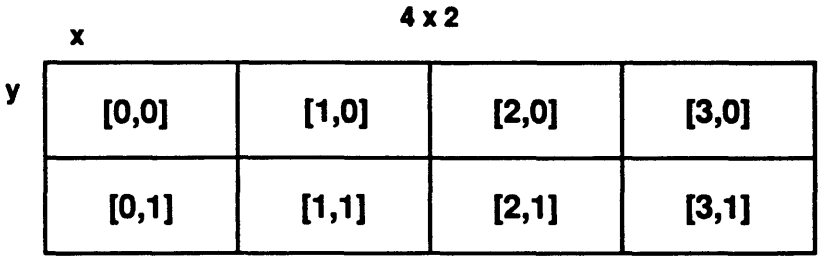
Because the array A was used in the array expression

```
A = 42
```

it is allocated on the PE array, and the OnDpu flag is set to TRUE.

Mapping Arrays onto the PE Grid

On the MasPar machine, arrays are mapped onto the PE grid in *columns* and *rows*. The first column of the first row is processor [0,0]. For instance:



How processors map on the machine

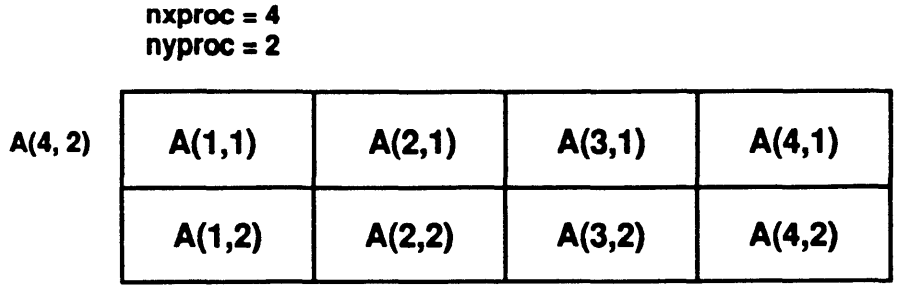
You can determine the number of PE rows and columns on your machine by executing the mpconfig command from the UNIX command line. This command prints the rows first, followed by the columns, which is the *opposite* of how rows and columns are referred to in MasPar Fortran two-dimensional array declarations.

Mapping MasPar Fortran Arrays

MasPar Fortran arrays are mapped onto the PE grid so that processor [0,0] contains the first Fortran array element. If the array is one dimensional, it is mapped in a serpentine fashion onto the PE grid. Assuming that the array size is smaller than the machine size, the Fortran element number maps directly onto the iproc number (that is, element 1 -> PE 0, element 2 -> PE 1, ...). (Mapping arrays that are larger than the machine is discussed later in this chapter.)

In MasPar Fortran, *always* think in terms of *columns* first, then *rows*.

A MasPar Fortran two-dimensional array that will be allocated on the PE grid is declared with the *first* dimension corresponding to the number of *columns* on the machine and the *second* dimension corresponding to the number of *rows*. For instance, in MasPar Fortran the array A(4,2) would be mapped onto the PE grid as follows:



How MasPar Fortran arrays map on the machine

The Active Set

In MPL, the **active set** (the processors taking part in the computation) consists of the entire machine, by default. In MasPar Fortran, the active set is defined by the size of the arrays in a Fortran 90 array expression. Unless the MasPar Fortran array is a two-dimensional array that corresponds exactly to the machine size, the MPL routine has to select the active set containing the array. If it does not select the active set, processors not containing array data would be active and an arithmetic exception could result. For example, in the following MasPar Fortran program, the array a occupies the processor grid [0:9, 0:19]. When the MPL program performs a computation on this array, only these processors take part in the computation.

MasPar Fortran:

```
      program mpltest
      implicit none
      integer, parameter :: ext1 = 10, ext2 = 20
      integer, dimension( ext1, ext2 ) :: a
      integer i, j

cmpf  mpl times_two

        a = 42
        call times_two( a )
        do j = 1, ext2
           print 100, (a(i, j), i = 1, ext1)
 100       format ( 1X, 10( I3 ))
        end do
      end
```

MPL:

```
#include <mpl.h>

typedef struct { plural int *addr;
                 unsigned int rank;
                 unsigned int OnDpu;
                 unsigned int extent[ 7 ];
               } DopeVector;


TIMES_TWO( fe_arg_blk, dpu_arg_blk )
      void *fe_arg_blk[];
      DopeVector   *dpu_arg_blk[];
{
  DopeVector *DopePtr;
  plural int *a;

  DopePtr = dpu_arg_blk[1];
  a = DopePtr->addr;
  if (ixproc < DopePtr->extent[ 0 ] && iyproc < DopePtr->extent[1])
    *a = *a * 2;
}
```

When compiled, linked, and executed, this code prints the following:

```
84 84 84 84 84 84 84 84 84 84
84 84 84 84 84 84 84 84 84 84
          [...]
84 84 84 84 84 84 84 84 84 84
```

Only the first two dimensions of an array passed to MPL are mapped onto the processor grid; the higher dimensions are mapped into PE memory. For example, the array

INTEGER A(2, 2, 100)

occupies a 2x2 processor grid. Each of these processors has 100 words of PE memory allocated for the third dimension.

The following example illustrates this mapping. The array a occupies three words in each processor. The MasPar Fortran main program numbers each of these, from 1 to 3. The MPL routine selects processor (0,0) and prints the values stored in the space allocated for the third dimension.

MasPar Fortran:

```
      program mpltest
      implicit none
      integer, parameter :: ext1 = 10, ext2 = 20, ext3 = 3
      integer, dimension( ext1, ext2, ext3 ) :: a
      integer i, j

cmpf  mpl print_layers

        do i = 1, ext3
           a(:, :, i) = i
        end do
        call print_layers( a )
      end
```

MPL:

```
#include <mpl.h>

typedef struct { plural int *addr;
                 unsigned int rank;
                 unsigned int OnDpu;
                 unsigned int extent[ 7 ];
               } DopeVector;


PRINT_LAYERS( fe_arg_blk, dpu_arg_blk )
      void *fe_arg_blk[];
      DopeVector   *dpu_arg_blk[];
{
  DopeVector *DopePtr;
  plural int *a;
  int i;

  DopePtr = dpu_arg_blk[1];
  a = DopePtr->addr;
  if (iproc == 0)
    for (i = 0; i < 3; i++) {
      printf("a(1,1,%d) = ", i+1);
      p_printf("%d\n", a[i]);
    }
}
```

When compiled, linked, and executed, this code prints the following:

```
a(1,1,1) = 1
a(1,1,2) = 2
a(1,1,3) = 3
```

Allocating Arrays that Are Larger than the Machine

MasPar Fortran supports computation on arrays that are sized larger than the machine size. To write MPL code that can perform computations on these large arrays, it is important to understand how they are mapped onto the machine.

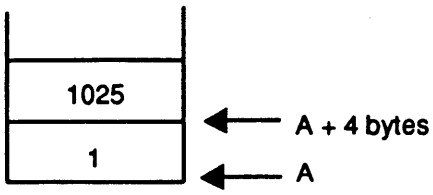
On a 1K processor machine, the array

INTEGER A(1024)

occupies one word of memory on each processor. The array

INTEGER A(2048)

occupies two words of memory on each processor. In this case, the first 1024 elements of the array are mapped into the first layer of A. The second 1024 elements are mapped into the second memory layer of A. If all the elements of A are numbered, starting with 1, the following results for PE 0:



Two-dimensional arrays are mapped using a technique known as **cut and stack**. See Figure 7-2.

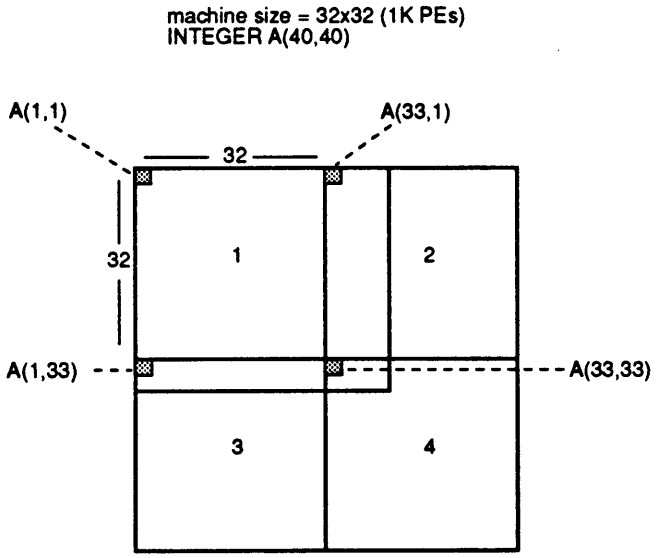


Figure 7-2 Mapping Two-Dimensional Arrays Using Cut and Stack

In Figure 7-2, the array

INTEGER A(40, 40)

is mapped onto a 32x32 (1K PEs) machine. The mapping is determined taking the smallest multiple of the machine size that will contain the array.

With this array, A(40, 40), this multiple is two in each dimension, yielding a 64x64 "virtual" machine. Because the machine multiple is two in each dimension, four 32-bit words are allocated on each processor for A.

If the array were A(70,40), a multiple of three in the first dimension and two in the second dimension would be used, yielding a virtual machine that is 96x64. In this case, then, there would be 3x2 = 6 words of memory on each PE allocated for **A**.

The **stacking** within the PE memory is shown by the numbering of the quadrants in Figure 7-2. The four words allocated for **A** in PE 0 will contain the shaded array elements in the diagram: A(1,1), A(33,1), A(1,33), and A(33,33). When called from a MasPar Fortran program that has been compiled for a 1K machine, the MPL code below prints out the four elements of **A** stored in PE 0. If your machine is larger than 1K, a program can still be compiled and run as if it were 1K by using the command-line option –pesize=1.

MPL:

```
#include <mpl.h>

typedef struct { plural int *addr;
                 unsigned int rank;
                 unsigned int OnDpu;
                 unsigned int extent[ 7 ];
               } DopeVector;


PRINT_STACKING( fe_arg_blk, dpu_arg_blk )
      void *fe_arg_blk[];
      DopeVector   *dpu_arg_blk[];
{
  DopeVector *DopePtr;
  plural int *a;
  int i;

  DopePtr = dpu_arg_blk[1];
  a = DopePtr->addr;
  if (iproc == 0)
    for (i = 0; i < 4; i++) {
      printf("a(%d) = ", i);
      p_printf("%d\n", a[i]);
    }
}
```

Figure 7-3 shows the processor ranges and memory offsets for each layer of the "virtual" machine.

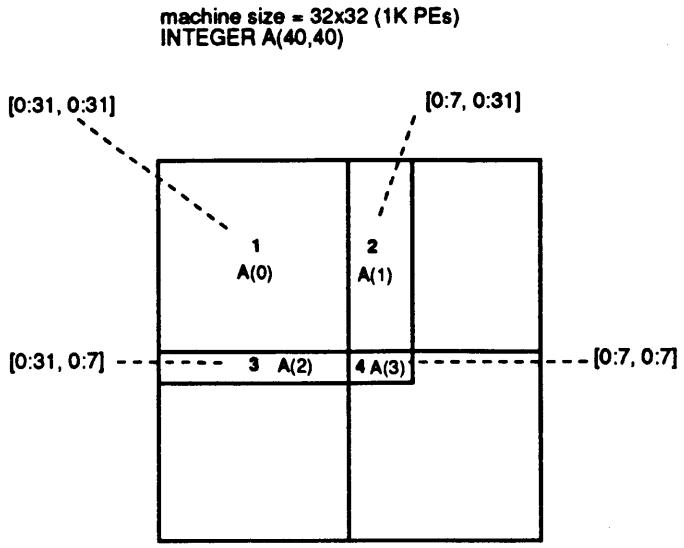


Figure 7-3 Layers of a "Virtual" Machine

The program that follows calls an MPL routine to number each of these memory layers according to its stacking level.

MasPar Fortran:

```
      program mpltest
      implicit none
      integer, parameter :: size = 40
      integer, dimension( size, size) :: a
      integer i, j

cmpf  mpl quad_number

        a = 0
        call quad_number( a )
        do j = 1, size, 4
           print 100, (a(i, j), i = 1, size, 4)
 100       format( 1X, 10( I2 ) )
        end do
      end
```

MPL:

```
#include <mpl.h>

typedef struct { plural int *addr;
                 unsigned int rank;
                 unsigned int OnDpu;
                 unsigned int extent[ 7 ];
               } DopeVector;


QUAD_NUMBER( fe_arg_blk, dpu_arg_blk )
      void *fe_arg_blk[];
      DopeVector   *dpu_arg_blk[];
{
  DopeVector *DopePtr;
  plural int *a;

  DopePtr = dpu_arg_blk[1];
  a = DopePtr->addr;
  if (ixproc < 32 && iyproc < 32)
    a[0] = 1;
  if (ixproc < 8 && iyproc < 32)
    a[1] = 2;
  if (ixproc < 32 && iyproc < 8)
    a[2] = 3;
  if (ixproc < 8 && iyproc < 8)
    a[3] = 4;
}
```

When compiled for a 1K machine and then linked and executed, this program prints the following:

```
1 1 1 1 1 1 1 1 2 2
1 1 1 1 1 1 1 1 2 2
1 1 1 1 1 1 1 1 2 2
1 1 1 1 1 1 1 1 2 2
1 1 1 1 1 1 1 1 2 2
1 1 1 1 1 1 1 1 2 2
1 1 1 1 1 1 1 1 2 2
1 1 1 1 1 1 1 1 2 2
3 3 3 3 3 3 3 3 4 4
3 3 3 3 3 3 3 3 4 4
```

The above example is coded with explicit knowledge of how the array (40x40) is mapped onto a machine of a specific size (32x32). An algorithm like this is of limited use, since it must be modified if it is to be transferred to a larger machine. A more generic algorithm would handle any array of rank two on a machine of any size. However, the above example demonstrates that mapping used in MasPar Fortran is complicated, and generic MPL routines that can handle any MasPar Fortran array of a specific rank are complex.

A less generic (but simpler) algorithm is one that processes arrays that are multiples of the machine size. The program that follows is given three rank-two DPU array

arguments: a, b, and c. It returns a with the result of b + c. These arrays must be **conformable** (that is, they must have the same dimensions). Note that when arrays are larger than the machine size (nxproc x nyproc) they will be *layered* through the PE memory. The i and j loops sequence through the layers of memory.

MPL:

```
#include <mpl.h>

typedef struct { plural int *addr;
                 unsigned int rank;
                 unsigned int OnDpu;
                 unsigned int extent[ 7 ];
               } DopeVector;

ADD( fe_arg_blk, dpu_arg_blk )
     void *fe_arg_blk[];
     DopeVector *dpu_arg_blk[];
{
  DopeVector *A_dope, *B_dope, *C_dope;
  int xlayers, ylayers, i, j, mem_offset;
  plural int *a, *b, *c;

  A_dope = dpu_arg_blk[ 1 ];
  B_dope = dpu_arg_blk[ 2 ];
  C_dope = dpu_arg_blk[ 3 ];
  a = A_dope->addr;
  b = B_dope->addr;
  c = C_dope->addr;
  xlayers = (A_dope->extent[ 0 ] + (nxproc - 1))/nxproc;
  ylayers = (A_dope->extent[ 1 ] + (nyproc - 1))/nyproc;
  for (j = 0; j < ylayers; j++)
    for (i = 0; i < xlayers; i++) {
      mem_offset = i + j * xlayers;
      a[ mem_offset ] = b[ mem_offset ] + c[ mem_offset ];
    }
}
```

The memory offset (mem_offset) is calculated using the array address polynomial. Given an array int a[e2][e1], the address in linear memory of the array element a[j][i] is i + j*e1. In the calculation of mem_offset, the number of elements per processor (xlayers) is used for the value of e1.

Following is an example of a MasPar Fortran routine that could be used to call ADD. Although the MPL code is more generic than this MasPar Fortran code, knowledge of the machine size is still embedded in the MasPar Fortran program, where the machine is defined as 64x32 (that is, 2K).

MasPar Fortran:

```
      program mpltest
      implicit none
      integer, parameter :: nxproc = 64, nyproc = 32
      integer, parameter :: xsize = nxproc * 2, ysize = nyproc * 2
      integer, dimension( xsize, ysize ) :: a, b, c

cmpf  mpl add

        a = 0
        b = 3
        c = 4
        call add( a, b, c )
      end
```

A version of ADD for three-dimensional arrays is shown below. When an array is passed to a subroutine, the first two dimensions of the array are mapped onto the PE grid, and the higher dimensions are mapped into memory. Therefore, on a 32x32 machine (1K PEs) the array

INTEGER A(64, 64, 4)

consumes six 32-bit words per PE (two words for the first two dimensions and four words for the last dimension). To calculate the memory offset, the address polynomial is used again:

int a[e3][e2][e1];

a[k][j][i] = a[i + j*e1 + k*e1*e2]

So the polynomial for computing the address is

i + j*xlayers + k*xlayers*ylayers

Factoring out xlayers produces

i + (j + k*ylayers)*xlayers

Note that the subexpression (j + k*ylayers)*xlayers is loop-invariant for the inner i loop and has been moved out of the loop.

MPL:

```
#include <mpl.h>

typedef struct { plural int *addr;
                 unsigned int rank;
                 unsigned int OnDpu;
                 unsigned int extent[ 7 ];
               } DopeVector;

ADD( fe_arg_blk, dpu_arg_blk )
     void *fe_arg_blk[];
     DopeVector *dpu_arg_blk[];
{
  DopeVector *A_dope, *B_dope, *C_dope;
  int xlayers, ylayers, i, j, k, mem_offset, loop_inv;
  plural int *a, *b, *c;

  A_dope = dpu_arg_blk[ 1 ];
  B_dope = dpu_arg_blk[ 2 ];
  C_dope = dpu_arg_blk[ 3 ];
  a = A_dope->addr;
  b = B_dope->addr;
  c = C_dope->addr;
  xlayers = (A_dope->extent[ 0 ] + (nxproc - 1))/nxproc;
  ylayers = (A_dope->extent[ 1 ] + (nyproc - 1))/nyproc;
  for (k = 0; k < A_dope->extent[ 2 ]; k++)
    for (j = 0; j < ylayers; j++) {
      loop_inv = (j + k * ylayers) * xlayers;
      for (i = 0; i < xlayers; i++) {
        mem_offset = i + loop_inv;
        a[ mem_offset ] = b[ mem_offset ] + c[ mem_offset ];
      }
    }
}
```

The preceding two examples assume that the arrays processed are multiples of the machine size. The example that follows assigns 42 to an integer array of rank two. The size of this array is not limited to being a multiple of the machine size.

MPL:

```
#include <mpl.h>

typedef struct { plural int *addr;
                 unsigned int rank;
                 unsigned int OnDpu;
                 unsigned int extent[ 7 ];
               } DopeVector;

ASSIGN( fe_arg_blk, dpu_arg_blk )
     void *fe_arg_blk[];
     DopeVector *dpu_arg_blk[];
{
  DopeVector *DopePtr;
  int xlayers, ylayers, ext_x, ext_y, i, j, xlim, ylim;
  plural int *a;

  DopePtr = dpu_arg_blk[ 1 ];
  a = DopePtr->addr;
  ext_x = DopePtr->extent[ 0 ];
  ext_y = DopePtr->extent[ 1 ];
  xlayers = (ext_x + (nxproc - 1))/nxproc;
  ylayers = (ext_y + (nyproc - 1))/nyproc;
  xlim = xlayers - ((ext_x & (nxproc-1)) != 0);
  ylim = ylayers - ((ext_y & (nyproc-1)) != 0);
  for (j = 0; j < ylim; j++) {
    for (i = 0; i < xlim; i++) {
      a[ i + j * xlayers ] = 42;                        /* section 1 */
    }
    if (ixproc < (ext_x & (nxproc-1)))
      a[ (xlayers - 1) + j * xlayers ] = 42;            /* section 2 */
  }
  if (iyproc < (ext_y & (nyproc - 1))) {
    for (i = 0; i < xlayers - 1; i++)
      a[ i + (ylayers-1)*xlayers ] = 42;                /* section 3 */
    if (ixproc < (ext_x & (nxproc-1)))
      a[ (xlayers-1) + (ylayers-1)*xlayers ] = 42;      /* section 4 */
  }
}
```

If ASSIGN is passed a 40x40 integer array and the machine has a 32x32 processor grid, the results are as follows:

```
ext_x   = 40
ext_y   = 40
xlayers =  2
ylayers =  2
nxproc  = 32
nyproc  = 32
(ext_x & (nxproc-1) = (0x28 & 0x1f) = 8
```

Figure 7-4 shows how the 40x40 array would be mapped onto a logical machine composed of 64x64 processors. In fact, this machine is composed of four 32x32 real machines, where each virtual machine is a layer of memory.

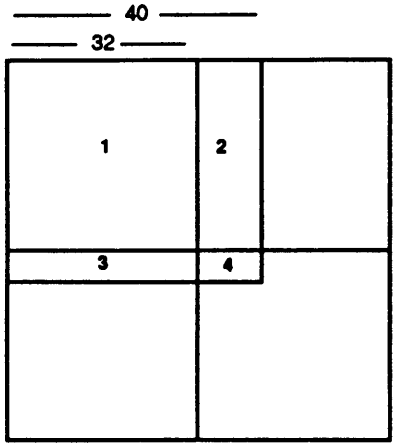
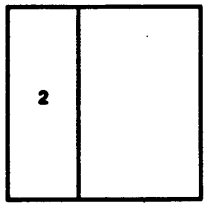


Figure 7-4 Mapping a 40x40 Array

The first loop in ASSIGN handles assignment to the first two sections in Figure 7-4. This loop is repeated below. Note that the active set for the inner i loop consists of the entire machine (section 1).

```
for (j = 0; j < ylim; j++) {
  for (i = 0; i < xlim; i++) {
    a[ i + j * xlayers ] = 42;                    /* section 1 */
  }
  if (ixproc < (ext_x & (nxproc-1)))
    a[ (xlayers - 1) + j * xlayers ] = 42;        /* section 2 */
}
```

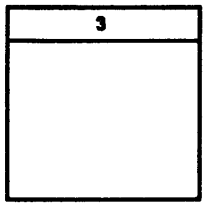
The active set for the if statement is the section [0..7, 0..31] diagramed here (see also Figure 7-4):



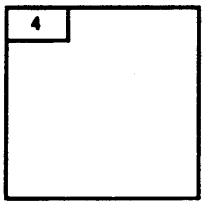
The second block of code, repeated below, handles sections 3 and 4 shown in Figure 7-4.

```
if (iyproc < (ext_y & (nyproc - 1))) {
  for (i = 0; i < xlayers - 1; i++)
    a[ i + (ylayers-1)*xlayers ] = 42;              /* section 3 */
  if (ixproc < (ext_x & (nxproc-1)))
    a[ (xlayers-1) + (ylayers-1)*xlayers ] = 42;    /* section 4 */
}
```

The active set for the inner i loop consists of processors [0..31, 0..7], which is section 3, shown below (see also Figure 7-4):



The active set for the final if statement consists of the active set [0..7, 0..7], section 4, shown here (see also Figure 7-4):



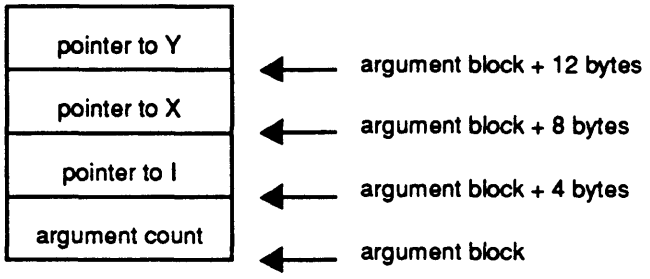
Accessing Front-End Scalar Arguments from MPL

Front-end scalars can also be accessed from an MPL subroutine. For example, the MasPar Fortran program below passes three scalars to an MPL subroutine named scalar_args.

MasPar Fortran:

```
      program mpltest
      implicit none
      integer i
      real x
      real*8 y

cmpf  mpl scalar_args

        i = 42
        x = 1.234E02
        y = 6.02D23
        call scalar_args( i, x, y )
      end
```

The addresses of these scalar values are passed to the MPL routine in an argument block that is in the front-end memory. The argument block is diagramed below:



Because the argument block and the values pointed to by the arguments are all on the front end, they must be fetched in a two-step process using the MPL library routine copyIn (see copyIn(3) in the *MPL Reference Manual*):

1. Copy the pointer from the front-end argument block into a variable on the ACU.
2. Copy the front-end value pointed to in step 1 into a variable on the ACU.

See Figure 7-5.

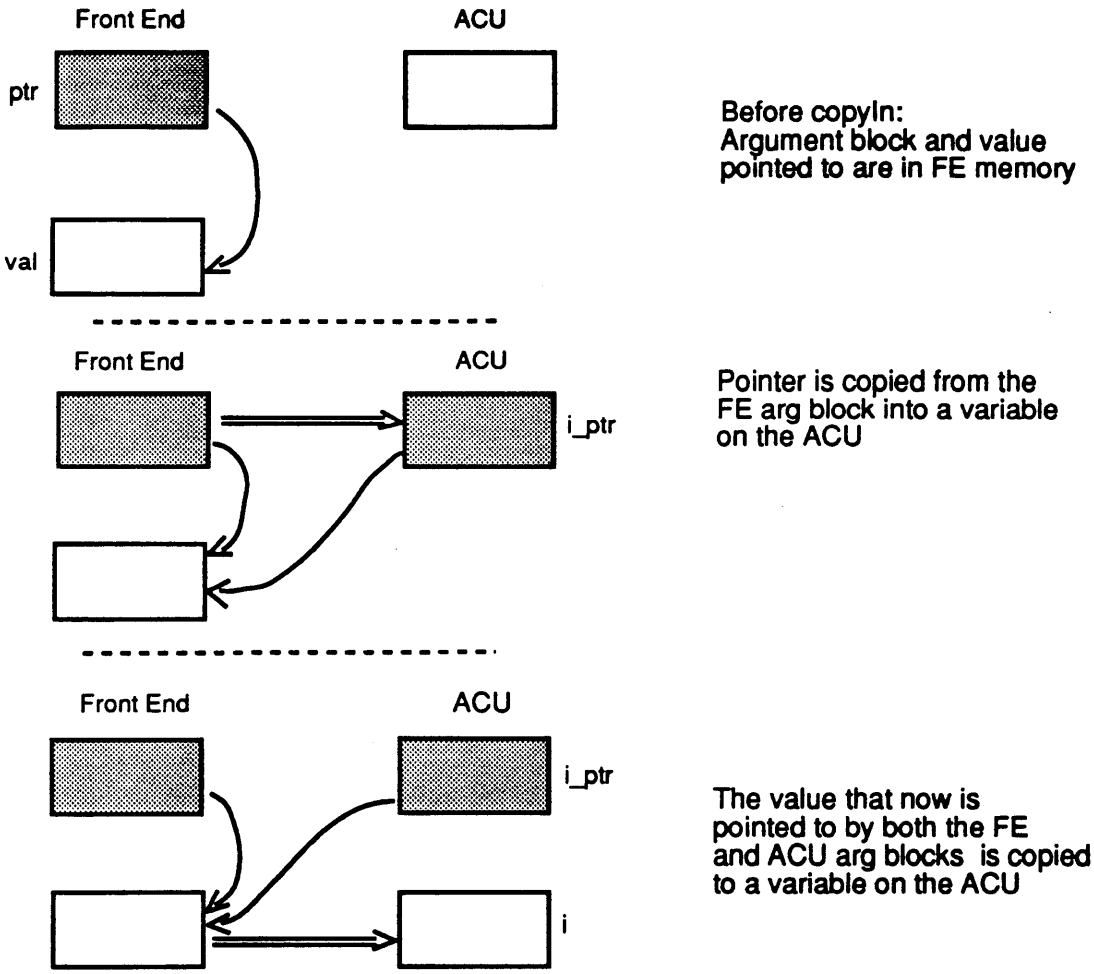


Figure 7-5 Copying Front-End Values to the ACU

This process is shown in the following MPL example. The fe_arg_blk argument is the address of the front-end argument block in front-end memory. Note that fe_arg_blk is defined as a void pointer array. The void type is used to emphasize that ACU operations on this variable are forbidden, since it contains a front-end address.

MPL:

```
typedef struct { plural int *addr;
                 unsigned int rank;
                 unsigned int OnDpu;
                 unsigned int extent[ 7 ];
               } DopeVector;

SCALAR_ARGS( fe_arg_blk, dpu_arg_blk )
      void *fe_arg_blk[];
      DopeVector *dpu_arg_blk[];
{
  int *i_ptr, i;
  float *x_ptr, x;
  double *y_ptr, y;

  /* get the addresses of the arguments */
  copyIn(&fe_arg_blk[1], &i_ptr, sizeof( i_ptr ));
  copyIn(&fe_arg_blk[2], &x_ptr, sizeof( x_ptr ));
  copyIn(&fe_arg_blk[3], &y_ptr, sizeof( y_ptr ));

  /* get the argument values */
  copyIn(i_ptr, &i, sizeof( i ));
  copyIn(x_ptr, &x, sizeof( x ));
  copyIn(y_ptr, &y, sizeof( y ));

  printf("i = %d\n", i );
  printf("x = %e\n", x );
  printf("y = %e\n", y );
}
```

When compiled, linked and executed, this code prints the following:

```
i = 42
x = 1.234000e+02
y = 6.020000e+23
```

Scalar values can be passed from MPL to the front end with the copyOut call (see copyOut(3) in the *MPL Reference Manual*). For example, the following MPL subroutine reads in the front-end argument i, multiplies it by two, and returns the new value to the front end, overwriting the old value.

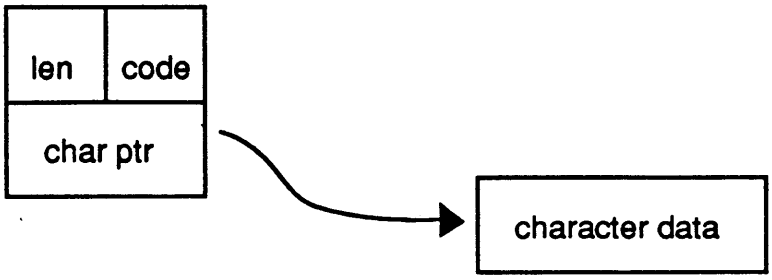
MPL:

```
typedef struct { plural int *addr;
                 unsigned int rank;
                 unsigned int OnDpu;
                 unsigned int extent[ 7 ];
               } DopeVector;

SCALAR_TIMES_TWO( fe_arg_blk, dpu_arg_blk )
     void *fe_arg_blk[];
     DopeVector *dpu_arg_blk[];
{
  int *i_ptr, i;

  copyIn(&fe_arg_blk[1], &i_ptr, sizeof( i_ptr ));
  copyIn(i_ptr, &i, sizeof( i ));
  i = i * 2;
  copyOut( &i, i_ptr, sizeof( i ));
}
```

If the scalar value passed to the MPL routine was originally a constant (instead of a variable), then its value cannot be altered. Any attempt to use this argument in a copyOut call will result in a front-end segment violation. This error will take place because the storage for constants is allocated in the "text" segment of the code file, which is read-only.

Accessing MasPar Fortran Character Variables

Character variables in MasPar Fortran differ in several ways from character strings in MPL. When a character value (a quoted character string or a character variable) is used as an argument to a subroutine the compiler creates a descriptor that is passed to the routine. This descriptor contains the length of the character data, a code used by the I / O system and a pointer to the actual string data. The MasPar Fortran character descriptor is shown here:



In MasPar Fortran character variables are always fixed length. If more character data is assigned to the character variable than will fit, the data is truncated. If the data is shorter, the remaining storage is blank-filled to the length of the variable. Fortran character variables are never null-terminated like strings are in MPL. For example, after the assignment below the character variable 'ch' will contain the characters "123", followed by seven blank characters.

```
character*10 ch

ch = '123'
```

The MPL routine PRINT_MPF_STRING in the next example demonstrates how a MasPar Fortran character variable can be accessed from MPL. The access must be done in two steps:

1. The Fortran character descriptor is read from front-end memory.
2. The character data is read from front-end memory, using the pointer address in the character descriptor.

Because printf expects a null-terminated string, care must be taken to null-terminate the character data before it is passed to printf.

MasPar Fortran:

```
      program test
      character*40 str
cmpf  mpl print_mpf_string

        str = "T'was Brillig and slithey toethes"
        call print_mpf_string( str )
      end
```

MPL:

```
typedef struct { short int len;
                 short int code;
                 char *str;
               } str_desc_type;

#define BUFSIZE 128

PRINT_MPF_STRING( fe_arg_blk, dpu_arg_blk )
     void *fe_arg_blk[];
     DopeVector *dpu_arg_blk[];
{
  int *str_desc_ptr;
  str_desc_type str_desc;
  char buf[ 128 ];
  int *len_ptr, len;

  /* Get the pointer to the front end string descriptor */
  copyIn( &fe_arg_blk[ 1 ], &str_desc_ptr, sizeof( int  ));
  /* Get the string descriptor */
  copyIn( str_desc_ptr, &str_desc, sizeof( str_desc_type ));
  printf("len = %d\n", str_desc.len );
  /* Get the character string */
  copyIn( str_desc.str, buf, str_desc.len );
  /* null terminate the string */
  if (str_desc.len < BUFSIZE)
    buf[ str_desc.len ] = '\0';
  else
    buf[ BUFSIZE-1 ] = '\0';
  printf("[%s]\n", buf );
}
```

Appendix A

MasPar Fortran Terms

Most of the new concepts introduced with MasPar Fortran relate to the use and processing of arrays.

array constructors	An array constructor is a sequence of scalar values interpreted as a one-dimensional array. The array element values are those specified in the sequence, as if an assignment were made for each element. The sequence of values can be specified as individual scalar values, ranges of values, or a one-dimensional array.
array inquiry functions	Array inquiry functions dynamically provide information on the size, bound, shape, etc. of a given array at runtime.
array-valued functions	Functions can return arrays of any numeric or logical data type (e.g., REAL, INTEGER, etc.).
assumed-shape arrays	Assumed-shape arrays allow you to avoid declaring the actual shape of the array in the called function. The MPF compiler automatically sets the shape correctly, based on the called function and the shape of the array being passed to it.
automatic arrays	Automatic arrays are dynamically allocated at runtime. Because these arrays do not have to be allocated when the program is started up, memory usage is more efficient.
vector-valued subscripts	One array can be constructed from another by the use of a vector that maps the desired elements into the target array. These subscripts are the mapping vector; they are called vector-valued subscripts.

These and other MasPar Fortran terms are discussed more fully in the *MasPar Fortran Reference Manual*.

Appendix B

Example Array Operations

This appendix teaches syntax and usage by providing examples of array operations in MasPar Fortran. These examples are broken down into the following five basic categories that cover some of the more common MasPar Fortran operations.

- simple arithmetic statements
- conditional expressions in parallel
- operations that move arrays across the PE grid
- reduction operations—operations that reduce an array to a scalar or to an array of lower rank
- new Fortran 90 intrinsic functions that are useful in linear algebra computations

Arithmetic Operations

In the following example, the subroutine of most interest is named XYZ. The Fortran 77 code is provided here. It is followed by the main loop of XYZ, alternatively shown in array syntax.

```
      program example_1

      parameter ( n  = 225 )
      parameter ( n1 = 224 )
      parameter ( a  = 1.0e-6 )
      parameter ( r1 = 0.001 )

      real r( 225,9,2 ), f( 225,9,2 ), q( 225,9 )

      num_iterations = 40

      do k = 1,num_iterations

         call xyz ( n1, a, r1, r, f, q, utot )

      end do

      stop
      end

      subroutine xyz ( n1, a, r1, r, f, q, utot )

      integer *4 n1

      real *4 r(225,9,2), f(225,9,2), q(225,9), a, utot

      real *4 dx(225,9,2), t1(225,9), t2(225,9), t3(225,9),
     &        x(225,9), x0(225,9), ue(225,9)

      utot = 0.0

      do j = 1,n1-1

         j1 = j + 1

         do i = 1,2
            do k = j1,n1
               dx( k,1,i ) = r( j,1,i ) - r( k,1,i )
               dx( k,2,i ) = r( j,1,i ) - r( k,2,i )
               dx( k,3,i ) = r( j,1,i ) - r( k,3,i )
               dx( k,4,i ) = r( j,2,i ) - r( k,4,i )
               dx( k,5,i ) = r( j,2,i ) - r( k,5,i )
               dx( k,6,i ) = r( j,2,i ) - r( k,6,i )
               dx( k,7,i ) = r( j,3,i ) - r( k,7,i )
               dx( k,8,i ) = r( j,3,i ) - r( k,8,i )
               dx( k,9,i ) = r( j,3,i ) - r( k,9,i )
            end do
         end do

         do m = 1,9
            do k = j1,n1
               t1(k,m) = dx(k,m,1)*dx(k,m,1) + dx(k,m,2)*dx(k,m,2)
               x (k,m) = 1.0 / t1(k,m)
               t2(k,m) = sqrt ( t1(k,m) )
```

```
            x0(k,m) = 1.0 / t2(k,m)
         end do
      end do

      do k = j1,n1
          ue(k,1) = q(j,1) * q(k,1)
          ue(k,2) = q(j,1) * q(k,2)
          ue(k,3) = q(j,1) * q(k,3)
          ue(k,4) = q(j,2) * q(k,4)
          ue(k,5) = q(j,2) * q(k,5)
          ue(k,6) = q(j,2) * q(k,6)
          ue(k,7) = q(j,3) * q(k,7)
          ue(k,8) = q(j,3) * q(k,8)
          ue(k,9) = q(j,3) * q(k,9)
      end do

      do m = 1,9
         do k = j1,n1
            t1(k,m)  = exp ( a * ( t2(k,m) - r1 ) )
            t2(k,m)  = 1.0 / t1(k,m)
            t3(k,m)  = t1(k,m) - t2(k,m)
            t2(k,m)  = t1(k,m) + t2(k,m)
            t3(k,m)  = t3(k,m) / t2(k,m)
            t2(k,m)  = 0.5 * t2(k,m)

            t1(k,m)  = ( 1.0 - t3(k,m) ) * ue(k,m) * x0(k,m)
            utot     = utot + t1(k,m)
            t2(k,m)  = ( t1(k,m) + ue(k,m) * a /
&                      (t2(k,m)*t2(k,m)) ) * x(k,m)
            f(k,m,1) = t2(k,m) * dx(k,m,1)
            f(k,m,2) = t2(k,m) * dx(k,m,2)
         end do
      end do

   end do

   return
   end
```

The loop below shows the code for the main loop in subroutine XYZ converted to Fortran 90 array syntax.

```
   do j = 1,n1-1

      j1 = j + 1

      dx( j1:n1,1:3,1 ) = r( j,1,1 )
      dx( j1:n1,4:6,1 ) = r( j,2,1 )
      dx( j1:n1,7:9,1 ) = r( j,3,1 )
      dx( j1:n1,1:3,2 ) = r( j,1,2 )
      dx( j1:n1,4:6,2 ) = r( j,2,2 )
      dx( j1:n1,7:9,2 ) = r( j,3,2 )

      dx(j1:n1,1:9,1:2) = dx(j1:n1,1:9,1:2) -
&                         r(j1:n1,1:9,1:2)

      t1(j1:n1,1:9)  = dx(j1:n1,1:9,1) * dx(j1:n1,1:9,1) +
&                      dx(j1:n1,1:9,2) * dx(j1:n1,1:9,2)
      x (j1:n1,1:9)  = 1.0 /  t1(j1:n1,1:9)
```

```
      t2(j1:n1,1:9)   = sqrt ( t1(j1:n1,1:9) )
      x0(j1:n1,1:9)   = 1.0 /  t2(j1:n1,1:9)

      ue(j1:n1,1:3)   = q(j,1)
      ue(j1:n1,4:6)   = q(j,2)
      ue(j1:n1,7:9)   = q(j,3)

      ue(j1:n1,1:9)   = ue(j1:n1,1:9) * q(j1:n1,1:9)

      t1(j1:n1,1:9)   = exp ( a * ( t2(j1:n1,1:9) - r1 ) )
      t2(j1:n1,1:9)   = 1.0 / t1(j1:n1,1:9)
      t3(j1:n1,1:9)   = t1(j1:n1,1:9) - t2(j1:n1,1:9)
      t2(j1:n1,1:9)   = t1(j1:n1,1:9) + t2(j1:n1,1:9)
      t3(j1:n1,1:9)   = t3(j1:n1,1:9) / t2(j1:n1,1:9)
      t2(j1:n1,1:9)   = 0.5 * t2(j1:n1,1:9)

      t1(j1:n1,1:9)   = ( 1.0 - t3(j1:n1,1:9) ) *
&                       ue(j1:n1,1:9) * x0(j1:n1,1:9)
      utot            = utot + sum ( t1(j1:n1,1:9) )
      t2(j1:n1,1:9)   = ( t1(j1:n1,1:9) + ue(j1:n1,1:9) * a
&                       /(t2(j1:n1,1:9) * t2(j1:n1,1:9)) ) *
&                       x(j1:n1,1:9)
      f(j1:n1,1:9,1) = t2(j1:n1,1:9) * dx(j1:n1,1:9,1)
      f(j1:n1,1:9,2) = t2(j1:n1,1:9) * dx(j1:n1,1:9,2)

end do
```

Note the conformance of scalars to arrays of all shapes (the code that uses the constants **a** and **r1**), as well as the fact that the intrinsic operations (EXP) accept array arguments. Additionally, scalar-to-array broadcasts occur when single array elements such as **r(j,1,1)** are assigned to entire array sections such as **dx(j1:n1,1:3,1)**. While the column section 1:9 is used in the above example, we could have just as easily taken the default, such as **t1(j1:n1,:)**.

This conversion illustrates only one method of conversion to array syntax. Study the following code segment:

```
      dx( j1:n1,1:3,1 ) = r( j,1,1 )
      dx( j1:n1,4:6,1 ) = r( j,2,1 )
      dx( j1:n1,7:9,1 ) = r( j,3,1 )
      dx( j1:n1,1:3,2 ) = r( j,1,2 )
      dx( j1:n1,4:6,2 ) = r( j,2,2 )
      dx( j1:n1,7:9,2 ) = r( j,3,2 )

      dx(j1:n1,1:9,1:2) = dx(j1:n1,1:9,1:2) -
&                         r(j1:n1,1:9,1:2)
```

Effectively, the conversion was performed in two steps:

1. initializing the appropriate column sections to a scalar value, and
2. performing the subtraction.

The code can actually be written in a more compact fashion, as shown below:

```
      dx( j1:n1,1:3,1 ) = r( j,1,1 ) - r( j1:n1,1:3,1 )
```

```
dx( j1:n1,4:6,1 ) = r( j,2,1 ) - r( j1:n1,4:6,1 )
dx( j1:n1,7:9,1 ) = r( j,3,1 ) - r( j1:n1,7:9,1 )
dx( j1:n1,1:3,2 ) = r( j,1,2 ) - r( j1:n1,1:3,2 )
dx( j1:n1,4:6,2 ) = r( j,2,2 ) - r( j1:n1,4:6,2 )
dx( j1:n1,7:9,2 ) = r( j,3,2 ) - r( j1:n1,7:9,2 )
```

In this version, the load of r involves the same active set as the store of dx. The scalar broadcast is followed directly by a subtract and a store. This is more efficient than the previous version, which stored dx away and then reloaded it in the next set of array statements.

In the above examples, the conversion was a relatively straightforward conversion of Fortran 77-style induction variables to Fortran 90-style array syntax.

You should be careful not to make array syntax substitutions everywhere there is an Fortran 77 loop. A classic example is the familiar first-order linear recurrence:

```
do i = 2,n
   a(i) = a(i-1) + c*b(i)
end do
```

This should *not* be replaced with the Fortran 90 syntax:

```
a(2:n) = a(1:n-1) + c*b(2:n)
```

To see the differences, type in both expressions in a test program, compile it, and observe the results. If the above array syntax does not conform to the Fortran 77, what Fortran 77 loop is it equivalent to? See the following example program:

```
program recurrence

parameter ( n = 10 )
real, dimension ( n ) :: a, b, c

do i = 1,n
   a(i) = i
   b(i) = i+1
   c(i) = i-1
end do

do i = 2,n
   a(i) = a(i-1) + b(i)*c(i)
end do

do i = 1,n
   print *, i, a(i)
end do

do i = 1,n
   a(i) = i
end do

a(2:n) = a(1:n-1) + b(2:n)*c(2:n)

print *, ' '
```

```
print *, ' '

do i = 1,n
   print *, i, a(i)
end do

do i = 1,n
   a(i) = i
end do

do i = n,2,-1
   a(i) = a(i-1) + b(i)*c(i)
end do

print *, ' '
print *, ' '

do i = 1,n
   print *, i, a(i)
end do

stop
end
```

The output of this program is as follows:

```
 1   1.000000
 2   4.000000
 3   12.00000
 4   27.00000
 5   51.00000
 6   86.00000
 7   134.0000
 8   197.0000
 9   277.0000
10   376.0000


 1   1.000000
 2   4.000000
 3   10.00000
 4   18.00000
 5   28.00000
 6   40.00000
 7   54.00000
 8   70.00000
 9   88.00000
10   108.0000


 1   1.000000
 2   4.000000
 3   10.00000
 4   18.00000
 5   28.00000
 6   40.00000
```

```
 7    54.00000
 8    70.00000
 9    88.00000
10    108.0000
```

Conditional Expressions in MasPar Fortran

In Fortran 90, parallel conditional statements use the WHERE construct. (See the *MasPar Fortran Reference Manual* for detailed information.) The following example demonstrates converting a Fortran 77 DO loop to a Fortran 90 WHERE construct.

Fortran 77 DO loops:

```
DO M = 1,9
   DO K = J1,N1
      IF ( ABS ( DX(K,M,1) ) .GT. HL ) THEN
         DX(K,M,1) = DX(K,M,1) - SIGN( CL,DX(K,M,1) )
      END IF
      IF ( ABS( DX(K,M,2) ) .GT. HL ) THEN
         DX(K,M,2) = DX(K,M,2) - SIGN( CL,DX(K,M,2) )
      END IF
      T1(K,M) = DX(K,M,1)*DX(K,M,1) + DX(K,M,2)*DX(K,M,2)
      X (K,M) = 1.0 / T1(K,M)
      T2(K,M) = SQRT ( T1(K,M) )
      X0(K,M) = 1.0 / T2(K,M)
    END DO
END DO
```

Note that since the same operations are performed on both planes of the DX array, we do not need two WHEREs. The following code shows the preceding Fortran 77 example converted to MasPar Fortran array syntax. The column dimension of the arrays is 9, and all row dimensions of the arrays are equal.

MasPar Fortran array syntax:

```
where ( abs ( dx(j1:n1,:,:) ) .gt. hl )
   dx(j1:n1,:,:) = dx(j1:n1,:,:) - sign( c1,dx(j1:n1,:,:) )
end where
t1( j1:n1,: ) = dx( j1:n1,:,1 ) * dx( j1:n1,:,1 ) +
&               dx( j1:n1,:,2 ) * dx( j1:n1,:,2 )
x ( j1:n1,: ) = 1.0 / t1( j1:n1,: )
t2( j1:n1,: ) = sqrt ( t1( j1:n1, : ) )
x0( j1:n1,: ) = 1.0 / t2( j1:n1,: )
```

In the assignment to T1, all arrays are aligned. The arrays DX and T1 are aligned in the first two dimensions. Since MasPar Fortran array allocation layers the third plane into PMEM, the addition of elements in the second plane causes no PE-PE communication. The ELSE WHERE part of the construct is shown below:

Fortran 77:

```
do j = 1,n
   do i = 1,n
      if ( a(i,j) .ne. 0.0 ) then
         a(i,j) = c / a(i,j)
      else
         a(i,j) = c
      end if
   end do
end do
```

MasPar Fortran array syntax:

```
where ( a(1:n,1:n) .ne. 0.0 )
   a(1:n,1:n) = c / a(1:n,1:n)
else where
   a(1:n,1:n) = c
end where
```

Elemental intrinsic functions can be used inside a WHERE construct. The masking applies to the elemental function, except when it is used as an argument to a non-elemental function. For example:

```
where ( a .ge. 0.0 )
   x = 2.0 * sum ( sqrt(a) )
end where
```

The above example causes the square root of all elements of a to be computed and then summed into a scalar temporary. That temporary is multiplied by 2.0 and stored into the scalar x. However, if some elements of a are less than zero, a runtime error occurs. On the other hand,

```
b = 0.0
where ( a .ge. 0.0 )
   b = sqrt( a )
end where
x = 2.0 * sum( b )
```

is a possible workaround. Since the array statement causes masking to be applied to a (an argument of an elemental function), no SQRT evaluations are made where a is negative.

Only array operations can be used inside a WHERE construct, and WHEREs cannot be nested. For example, the following code is invalid:

```
where ( a .ge. 0.0 )
   b = sqrt( a )
else where
   print *, 'Negative elements in a ...'
end where
```

This implies that a scalar IF cannot be used inside a WHERE construct; however, the following code is legal:

```
real *4 a(100,100,10), b(100,100,10), c(100), x

do k = 1,10
   if ( c(k) .ne. 0.0 ) then
```

```
      where ( a(:,:,k) .ne. 0.0 )
         a( :,:,k ) = b( :,:,k ) / a( :,:,k )
         a( :,:,k ) = a( :,:,k ) * c(k)
      end where
   end if
end do
```

In the above code, the scalar IF is used as a control statement. While it is not necessarily the best way to write the code, it does illustrate that you can have WHEREs inside IFs but not IFs inside WHEREs.

To handle nested IF statements in a parallel context, break the nested IF statements into discrete blocks containing slightly more complicated logical operations. A simple example is provided in the following code:

```
      program where_test

      parameter ( n = 100 )

      real, dimension ( n,n ) :: a, b, c, a1, b1, c1
      real x, sum1, sum2
c
c     A simple test program to illustrate how to convert nested IF
c     statements to WHEREs.
c
      do j = 1,n
         do i = 1,n
            a(i,j) = 0.5 * ( i + j )
            b(i,j) = i
            c(i,j) = 0.0
         end do
      end do

      a1 = a
      b1 = b
      c1 = c
      x  = n / 4.0

      do j = 1,n
         do i = 1,n
            if ( a(i,j) .le. x ) then
               if ( b(i,j) .ge. x ) then
                  c(i,j) = a(i,j) + b(i,j)
               else
                  c(i,j) = a(i,j) - b(i,j)
               end if
            end if
         end do
      end do

      sum1 = sum ( c )

      where ( ( a1 .le. x )  .and.  ( b1 .ge. x ) )
         c1 = a1 + b1
      end where

      where ( ( a1 .le. x )  .and.  ( b1 .lt. x ) )
         c1 = a1 - b1
      end where

      sum2 = sum ( c1 )
```

```
print *, sum1, sum2

stop
end
```

The compiler translates WHERE statements into instructions that set the execution bit (EBIT) on each PE. PEs whose EBIT is turned ON participate in the array operations for however many layers are required.

Array Movement Operations

This section takes a brief look at Fortran 90 intrinsic functions for shifting and spreading arrays. Other intrinsics exist for moving and shaping/reshaping arrays. For complete descriptions of all the intrinsics available with MasPar Fortran, see the *MasPar Fortran Reference Manual.*

EOSHIFT and CSHIFT Intrinsic Functions

The EOSHIFT (end-off shift) intrinsic function moves array elements in a specified direction for a specified distance. Values that move beyond array bounds "fall off" the end of the array, and a default boundary value(s) is inserted at the other end. The following program illustrates EOSHIFT syntax.

```
program eoshift_test

real, dimension(10) :: a

do i = 1,10
   a(i) = i
end do

a = eoshift ( a, dim = 1, shift = 1 )

do i = 1,10
   print *, i, a(i)
end do

print *, ' '

a = eoshift ( a, dim = 1, shift = -4 )

do i = 1,10
   print *, i, a(i)
end do

stop
end
```

In the output of this program shown below, notice how a positive or negative value of the SHIFT argument is used to control the direction of the shift. The default value of 0.0 for BOUNDARY is used in this example.

```
 1   2.000000
 2   3.000000
 3   4.000000
 4   5.000000
 5   6.000000
 6   7.000000
 7   8.000000
 8   9.000000
 9   10.00000
10   0.0000000E+00

 1   0.0000000E+00
 2   0.0000000E+00
 3   0.0000000E+00
 4   0.0000000E+00
 5   2.000000
 6   3.000000
 7   4.000000
 8   5.000000
 9   6.000000
10   7.000000
```

The array a is used in a scalar context in three instances and in a parallel context in two instances. It moves from the front end to the DPU twice (before calls to EOSHIFT) and from the DPU to the front end in two instances (one element at a time in each of the print loops). The initialization is performed on the front end, because a is used in a scalar context in the Fortran 77 loop.

In contrast to EOSHIFT, the CSHIFT intrinsic function moves the elements of an array in a circular shift; elements shifted out at one end are shifted in at the other end. Following is the same code example as above using CSHIFT instead of EOSHIFT:

```
program cshift_test

real, dimension(10) :: a

do i = 1,10
   a(i) = i
end do

a = cshift ( a, dim = 1, shift = 1 )

do i = 1,10
   print *, i, a(i)
end do

print *, ' '

a = cshift ( a, dim = 1, shift = -4 )

do i = 1,10
   print *, i, a(i)
end do

stop
end
```

```
 1   2.000000
 2   3.000000
 3   4.000000
 4   5.000000
 5   6.000000
 6   7.000000
 7   8.000000
 8   9.000000
 9   10.00000
10   1.000000

 1   8.000000
 2   9.000000
 3   10.00000
 4   1.000000
 5   2.000000
 6   3.000000
 7   4.000000
 8   5.000000
 9   6.000000
10   7.000000
```

This example shows circular shifts of a two-dimensional array in both dimensions. Note the use of the DIM argument with these intrinsics.

```
program circular_shift

parameter ( ndim = 5 )
real, dimension (ndim,ndim) :: a

do j = 1,ndim
  a( :, j ) = j
end do

a = cshift ( a, dim = 2, shift = 1 )

do i = 1,ndim
   print *, ( a(i,j), j = 1,ndim )
end do

do j = 1,ndim
  a( j, : ) = j
end do

print *, ' '

a(1:3,:) = cshift ( a(1:3,:), dim = 1, shift = 1 )

do i = 1,ndim
   print *, ( a(i,j), j = 1,ndim )
end do

stop
end

2.000000      3.000000      4.000000      5.000000      1.000000
2.000000      3.000000      4.000000      5.000000      1.000000
2.000000      3.000000      4.000000      5.000000      1.000000
```

```
2.000000      3.000000      4.000000      5.000000      1.000000
2.000000      3.000000      4.000000      5.000000      1.000000


2.000000      2.000000      2.000000      2.000000      2.000000
3.000000      3.000000      3.000000      3.000000      3.000000
1.000000      1.000000      1.000000      1.000000      1.000000
4.000000      4.000000      4.000000      4.000000      4.000000
5.000000      5.000000      5.000000      5.000000      5.000000
```

SPREAD Intrinsic

The SPREAD intrinsic expands an array by making copies of it and "spreading" these copies across the PE grid in a specified manner. A common analogy is making a book by "spreading" a single page into multiple pages, all alike. (See the *MasPar Fortran Reference Manual* for more information on SPREAD.) Examine the following example:

```
      program spread_test

      real, dimension (10,10) :: a
      real, dimension (10)    :: x

      a        = 0.0
      a( 1,: ) = 1.0
      x        = 1.0

      print *, ' '
      print *, 'Before ... '
      print *, ' '
      do i = 1,10
      write ( 5,10 ) ( a(i,j), j = 1,10 )
10      format ( 1x, 10f6.1 )
      end do

      a( 2:10, : ) = spread ( a(1,:), dim = 1, ncopies = 9 )

      print *, ' '
      print *, 'After ... '
      print *, ' '
      do i =1,10
         write ( 5,10 ) ( a(i,j), j = 1,10 )
      end do
c
c     Now, let's do it again ...
c
      a        = 0.0
      a( :,1 ) = 1.0

      print *, ' '
      print *, 'Before ... '
      print *, ' '
      do i = 1,10
         write ( 5,10 ) ( a(i,j), j = 1,10 )
      end do

      a( :, 2:10 ) = spread ( a(:,1), dim = 2, ncopies = 9 )
```

```
      print *, ' '
      print *, 'After ... '
      print *, ' '
      do i = 1,10
         write ( 5,10 ) ( a(i,j), j = 1,10 )
      end do
c
      a = a + spread ( x, dim = 2, ncopies = 10 )

      print *, ' '
      print *, 'After update... '
      print *, ' '
      do i = 1,10
         write ( 5,10 ) ( a(i,j), j = 1,10 )
      end do
c
      a( 5:10,5:10 ) = a( 5:10,5:10 ) -
     &                 spread ( a(4,5:10), dim = 1, ncopies = 6 )
*
     &                 spread ( a(5:10,4), dim = 2, ncopies = 6 )

      print *, ' '
      print *, 'After rank-1 update... '
      print *, ' '
      do i = 1,10
         write ( 5,10 ) ( a(i,j), j = 1,10 )
      end do
c
      a( 7:10,7:10 ) = a( 7:10,7:10 ) -
     &               spread ( a(5,7:10), dim = 1, ncopies = 4 ) *
     &               spread ( a(7:10,5), dim = 2, ncopies = 4 ) -
     &               spread ( a(6,7:10), dim = 1, ncopies = 4 ) *
     &               spread ( a(7:10,6), dim = 2, ncopies = 4 )

      print *, ' '
      print *, 'After rank-2 update... '
      print *, ' '
      do i = 1,10
         write ( 5,10 ) ( a(i,j), j = 1,10 )
      end do
c
      stop
      end
```

The output of this SPREAD_TEST program is:

Before ...

```
1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

After ...

1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Before ...

1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

After ...

1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0

After update ...

2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0
2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0