Writing Data Parallel Programs with High Performance Fortran

Student Notes

Version 1.2

A K Ewing, H Richardson,

A D Simpson, R Kulkarni

Edinburgh Parallel Computing Centre

The University of Edinburgh

Table of Contents

Course Timetable

Day 1

0900 Registration

0910 **Lecture 1**: Introduction to Data Parallel Programming

1000 Exercise 1: Introduction to the compiler

1030 Coffee

1100 **Lecture 2**: Fortran 90 Features

1200 Exercise 2: Game of Life

1300 Lunch

1400 **Lecture 3**: Basic Data Mapping

1420 Exercise 3: Life Revisited

1500 Tea

1530 **Lecture 4**: HPF Parallel Features

1600 Exercise 4: Mandelbrot Set

1700 Close

Day 2

0900 **Lecture 5**: Data Mapping

1045 Exercise 5: Birthday Exercise

1100 Coffee

1130 **Lecture 6:** Data Mapping and Procedure Arguments

1215 Exercise 6: Life in a subroutine

1300 Lunch

1400 **Lecture 7**: Intrinsic Functions and HPF Library

1445 Exercise 7: Golf Scores

1530 Tea

1600 **Lecture 8**: Advanced Topics and Current Developments

1630 End

1 Data Parallel Programming

1.1 Introduction

Much of the impetus behind the development of high performance computing, and in particular parallel computing, is the success of computational science in describing natural phenomena by means of numerical simulations. The desire for larger and more detailed simulations continues to fuel the demand for low cost, high performance computers.

During the evolution of the digital computer this demand has been partially met by advances in hardware and software technologies. However, the increases in performance of a single processor are ultimately limited by economic factors (the escalating cost of producing novel technologies) and by fundamental physical laws (information in a processor cannot travel faster than the speed of light, and the distance between information pathways is bounded below by the laws of quantum mechanics).

One promising approach to delivering this much sought after performance is that of parallel processing: the use of multiple processors working together to complete a task. To succeed with this approach it must be possible to break down a problem into smaller tasks or domains, with each processor working on a small part of the problem. State of the art parallel machines can achieve computation rates of 10^{11} floating point operations per second (flops), with the holy grail of high performance computing being an affordable "Teraflop" machine (10^{12} flops), still perhaps a decade away.

However, one of the main criticisms of parallel computing, and perhaps one which may limit its future, is the need to re-engineer software to run on a parallel machine. This results from the need for the programmer to explicitly tell the set of processors how to co-operate in solving the problem. Although a number of languages have been developed which target parallel computers, usually extensions of well-known conventional languages like Fortran, C and Pascal, it is still a massive job to convert large and often long evolved codes to run efficiently on parallel machines. This problem of porting codes to parallel machines is compounded by the differences between the languages and the machine specifics often included. For parallel computing to mature as a technology, there is clearly a need for a portable and widely-acceptable standard to emerge.

High Performance Fortran (HPF) is the result of efforts towards the standardisation of a Fortran language suitable for the latest generation of high performance machines. It is a set of constructs and extensions to Fortran90 and allows the user to express parallelism in a relatively simple manner. The main aims of such a standard are to promote the wider use of parallelism by hiding the details of the underlying architecture from the programmer and to provide a code which is easily portable and non-machine specific.

In subsequent sections we shall consider parallel architectures and introduce concepts relating to parallel computing. The data parallel programming style is introduced and placed in context with other approaches to the programming of parallel machines. We conclude this chapter with an outline of the building blocks required for a data parallel language and mention some of the array processing features of Fortran 90 which make it such a good starting point for HPF.

1.2 Parallel Architectures

Parallel computers vary greatly in complexity: a small machine may only have a handful of processors on one circuit board, a larger machine might have many thousands of complex processors connected by a number of specialised communications networks contained in many cabinets. As such, classifying these architectures in a sensible way is also problematic.

However, there exist simplistic models of parallel computer architectures which convey the important features. Basically, any parallel computer consists of three main elements: processors, memories and an interconnection network to enable communication between these elements. However, the wide range of architectures existing today arise from the way these elements are connected. A typical way to classify these architectures is with Flynn's taxonomy, which labels architectures according to instruction stream and data stream. For example, an idealised serial computer would be labelled Single Instruction Single Data (SISD) as it executes one instruction at a time on a single piece of data. In the following sections we outline the main classes of parallel architectures.

1.2.1 Single Instruction Multiple Data (SIMD)

The SIMD architecture consists of many (typically simple) processors, with some local memory, executing the same instruction in lockstep, on a small piece of data in its local memory, with the instructions issued by the controller processor. Such architectures are good for applying algorithms which require the same operation on a large array of elements, however, they suffer badly if the problem results in load imbalances as the processors synchronise after every step. This is not usually a problem for data parallel programs.

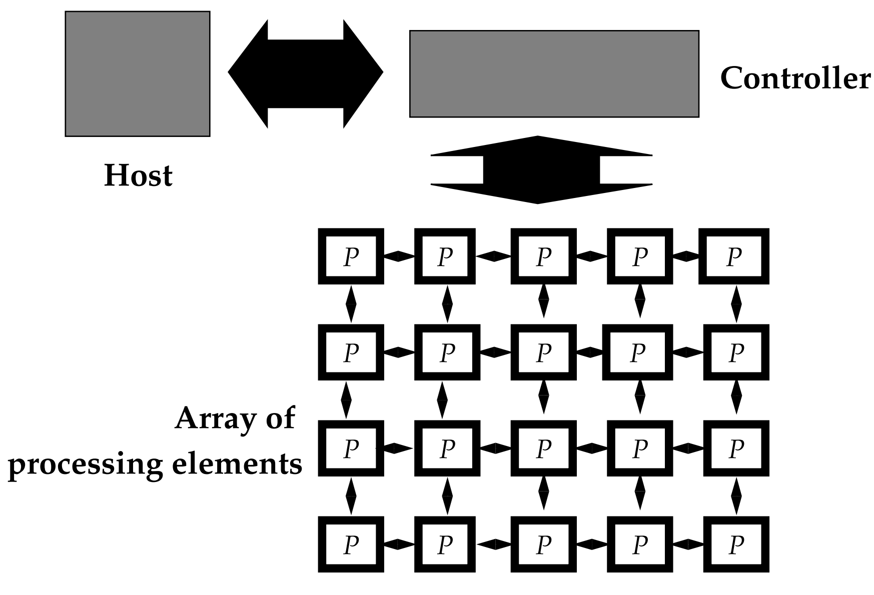


Figure 1: SIMD Architecture

1.2.2 Multiple Instruction Multiple Data (MIMD)

MIMD architectures consist of a number of powerful processors which can each execute individual instruction streams. The usual subdivision of this class is by the relationship between the processor and memory.

Distributed Memory

Figure 2 shows an example of a distributed memory system, where each processor has its own memory and each processor-memory pair is connected to an interconnect network. Processors can only read their local memory and cannot directly access the memory of another processor. In order to access remote data the processors must communicate via explicit message passing, programmed by the user. Clearly, access times differ depending on the memory accessed: local memory access quickest, with remote access slower and dependent on the physical separation of remote processors.Also, this architecture introduces a new level of technical knowledge required to program a MIMD computer, though this is often outweighed by the possible gains to be made from scalable performance possibilities.

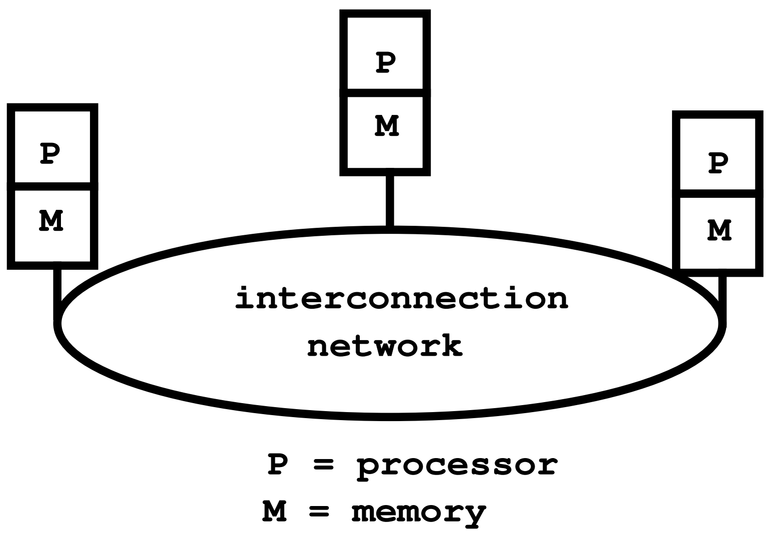


Figure 2: Distributed Memory Architecture

Shared Memory

For the architecture detailed in Figure 3, each processor is connected via the interconnect network to the memory elements. Hence each processor can access any part of the memory directly, with memory access times uniform for all processors. This is known as a shared memory architecture. These architectures are typically much easier to programme than distributed memory architectures, requiring no explicit message passing, however, they suffer from limited scalabilty where bottlenecks occur when multiple processors try to access the same memory location.

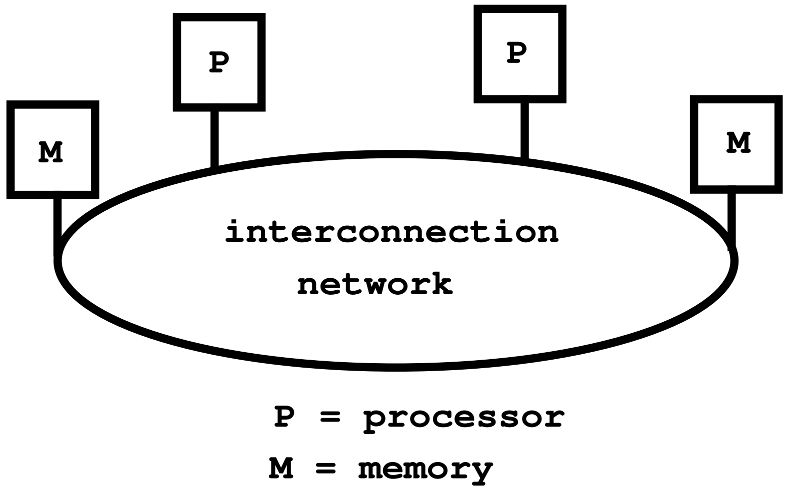


Figure 3: Shared Memory Architecture

Virtual Shared Memory

The problem with the above classifications is that they represent idealised architectures. In real machines a combination of these features is often utilised. One important class of these hybrids is the so called virtual shared memory architecture. Each processor has its own local memory (like a distributed machine), however, direct remote memory access (like a shared memory machine) is possible via a global address space. This relies on special hardware to deal with the communication, while the processors continue computing. Such a systems benefits from fast communications with a scalable architecture.

1.3 Programming Paradigms

In the previous section we outlined the various architectures often adopted for parallel machines. However, this is only part of the story, and takes no account of the programming models. At the most basic level, two main styles (or paradigms) of programming have emerged and gained acceptance by the user community, namely, *message passing* and *data parallel*. In this section we introduce these styles, however, in what follows we will primarily concentrate on the data parallel paradigm since it is of direct relevance to High Performance Fortran.

In both of these paradigms, the same basic issues of parallelism must be considered. Typically, on parallel machines the problem to be solved is divided up over the processors available, with the aim being to produce the same solution as using a serial machine. There are many ways of implementing this problem decomposition, the details of which do not concern us here. More importantly, the aim of choosing a suitable problem decomposition (for both the target architecture and algorithm utilised) is to provide an evenly balanced load, with every processor being kept busy, and also to minimise the inter-processor communications. The different programming paradigms provide different means of controlling these issues.

1.3.1 Message Passing

In the message passing paradigm a separate program is loaded onto each processor, typically on a MIMD machine. Each program is written in a standard language (C or Fortran 77, for example), with movement of data being controlled by calls to communication routines from some communication library. The programmer has complete control over the distribution of the data and communication between processors, with the responsibility to organise the processes so that they operate collectively.

To understand this more clearly, consider the following example: a set of processors with each processor programmed to be responsible for some of the elements of an array **`A`**. If each processor requires access to an element of **`A,`** say **`A(10)`**, in order to do some computation, only the processor with element **`A(10)`** in local memory can access it directly. All other processors have to call a communication routine to fetch the required element of **`A`**. This requires the programmer to explicitly specify; the message to be sent (data to be sent, address to be sent to etc.) and the corresponding receive call. In addition, sufficient cooperation between the send and receive is necessary to avoid deadlock.

If, continuing with this example, the maximum value in the array has to be found and all processors require this value. The programmer might go about this in the following way; find the maximum value of the array elements held in its local memory and then to communicate these (nodal) maxima to a single node where they are combined into a global maximum. The global maximum will have to be sent to all nodes. This again raises the issues of cooperation between sends and receives and also of the need to synchronise calculations. Often efficient implementations of much used functions, such as maximum or sum, are provided by the manufacturers within the communication library.

Communications clearly introduce additional overheads in the program execution, but they are essential parts of multiprocessor programming, often there being a trade-off between compute speed and communication. Communication may be required to move data as we have seen in the example above or to convey control messages so that tasks can be efficiently divided between processors. In both cases, this is done explicitly by the programmer.

The message passing style of programming is appropriate for distributed memory machines. It is also very flexible but has the following disadvantages:

- Each processor can be performing a different task and executing different code at any given instant. This makes the understanding and debugging of an application very difficult.
- The programmer has to ensure that all processors are kept busy as much as possible.
- The user has to be careful to minimise the number of messages and to attempt to overlap computation and communication whenever possible.
- It is important to make sure that an application does not ‘deadlock’. This can happen for example if messages are sent but not received, the result being that the application will hang.

In summary, the message passing paradigm gives the programmer a greater freedom in program control, but devloves to the programmer the full responsibility of making the program work and of keeping the processors busy and communicating smoothly.

1.3.2 Data Parallel Programming

The idea behind the data parallel programming paradigm is the support of whole array operations executed in parallel. Typically a single program controls the distribution of, and operations on the data on all processors. The languages used to program this vary from standard Fortran or C, with language extensions to deal with the parallelism, to specialised data parallel languages based on one machine. In most cases the actual distribution of data and communication between processors is done by the compiler, with guidance from the programmer.

A more definitive way to describe the data parallel programming style is through the use of the following headings:

- **single threaded control**: one program defining the operations
- **global namespace**: programmer only views a single memory, with all the low-level details of data distribution, memory access and communication dealt with by the compiler.
- **loosely synchronous:** execution of each instruction is not synchronised across all processors. However some program constructs, such as the completion of a loop, do force synchronisation. Although each processor executes the same piece of code, there is no guarantee that each processor is executing the same instruction at the same instant in time.
- **parallel operations**: operations on array elements executed simultaneously across all processors.

Underpinning all these descriptions is the concept of transferring the responsibility for the low level details of the programming from the programmer to the compiler, thus freeing the programmer to concentrate on the application.

This concept is further strengthened by the provision of routines for commonly used operations on which to base data parallel algorithms. Such routines, typically manufacturer's efficient implementations making the most of the host architecture, increased the programmers productivity by avoiding "re-inventing the wheel" and even simple syntax errors.

The aim of this focussing away from the machine/langauge specifics is to encourage the wider use of parallel computing. Typically this is aimed at applications programmers who wish to exploit the price/performance of parallelism but are unwilling to spend the time and effort porting their long developed code to a message passing environment, perhaps using a different programming language.

1.4 Data Parallel Building Blocks

In the last section, we described the two main programming paradigms: message passing and data parallel. This course describes only the data parallel paradigm via the use of High Performance Fortran (HPF). Before we go on to describe HPF in detail, we outline the types of operations a programmer is concerned with in applications suitable for the data parallel style. This is an idealised set of operations of which the manufacturers should provide efficient implentations.

Data parallel programming is concerned with defining collective operations on arrays or sets of array elements, with these arrays distributed over a number of processors. If an algorithm can be expressed in terms of such operations then it is likely that a data parallel implementation will be efficient. Grid based algorithms are one good example. It is helpful to categorise the set of operations that form the basis for the implementation of data parallel algorithms. These are:

- control over data layout
- whole array operations
- array sections
- conditional operations
- reduction operations
- shift operations
- scan operations
- generalised communications

We will now consider each of these in more detail.

1.4.1 Controlling Data Layout

In many cases it is crucial that the user has control over the placement of data on the processors. The goals are to minimise communication between processors, keep all processors busy and to carry out operations in parallel to obtain highest performance. Data layout is normally controlled by language constructs or directives. Figure 4 shows two possible data layouts for a two dimensional array. The shaded portion indicates array elements and how the data could be distributed across processors P1, P2, P3, P4.

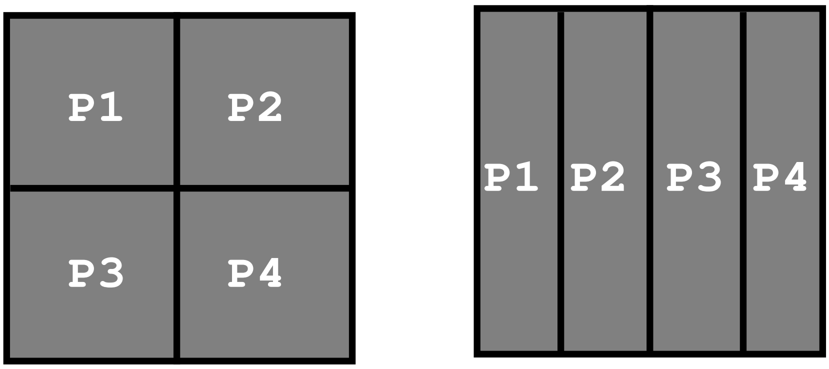


Figure 4: Typical data layouts

1.4.2 Whole Array Operations

Such operations would take as arguments whole arrays and apply the operation to every individual element of that array. The operation, such as sum, multiply and divide, is applied to each element of the array, possibly in parallel. Figure 5 shows the multiplication of two arrays. All the elements can be multiplied in parallel given a sufficient number of processors.

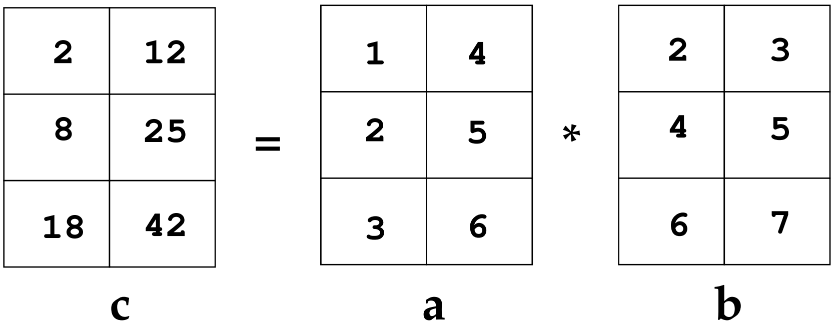


Figure 5: Whole array operation

An implementation should support array expressions and also extend the standard mathematical functions (*sin*, *cos* etc.) to operate on array-valued arguments, applying the operation to every array element and returning an array of results. The intent on of supplying such whole array operations is to enable the production of clearer code and to reduce the likelihood of mistakes.

1.4.3 Array Sections

There should be a method to access sections of an array. This would allow the programmer to specify regular sections of array on which to act. Figure 6 shows selection of a column of an array and selection of the interior elements of an array.

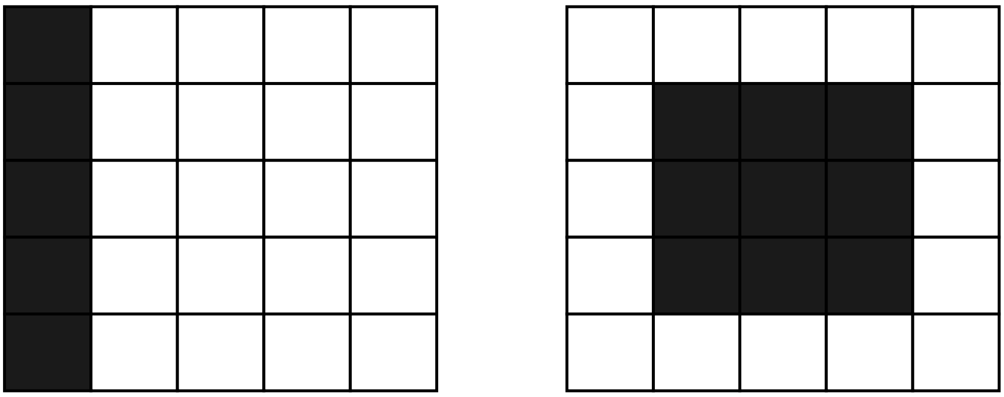


Figure 6: Array sections

This would be useful for, say multiplying matrices where whole rows and columns are multiplied, or the update of some grid based problem, where only the central array elements are of interest as opposed to the "halo" around the edge.

1.4.4 Conditional Operations

Operations can be made to act on a subset of array elements, chosen subject to some conditional based (logical) mask or expression. This would allow the programmer to specify irregular sections of the array on which to act. For example, Figure 7 shows the selection of elements of a 7x7 element array, a shaded square indicating that the mask is set in this position.

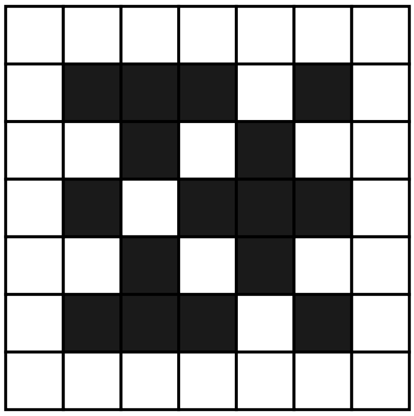


Figure 7: Conditional operations on arrays

This could be used to, say, precondition the grid problem and perform the update on the even and odd sites separately, or to apply an operation only to array elements which satisfy some condition, such as being not equal to zero.

1.4.5 Reduction Operations on Arrays

A reduction operation produces one result from the combination of many elements of an array. Examples of possible reduction operations are:

- sum of elements in array,
- minimum or maximum values in the array,
- logical AND, OR, EOR,
- a count of the number of true elements in a logical array.

These operations are useful in control constructs where the logical flow of the program depends on some global property of an array. An example would be a converging iterative procedure applied to all elements of an array or array section. The iteration process could stop when all processors had achieved some tolerance, in other words when all values in some logical mask expression were true.

1.4.6 Shift Operations on Arrays

Many efficient parallel algorithms can be implemented in terms of shifts along axes of multidimensional arrays, notable examples being image processing algorithms, finite difference methods and cellular automata simulations.

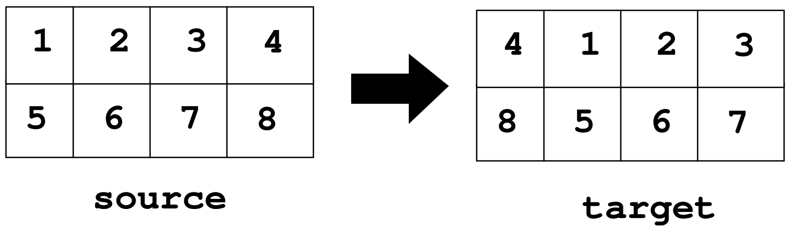


Figure 8: Shift operations

In Figure 8, the elements of a two dimensional array are shifted by one position along axis 2. This shift operation allows the edge elements to “wrap around” the array when shifted off the end of the array. In addition, “end-off” shifts are also possible.

1.4.7 Scan Operations

Scan or prefix/suffix operations the elements of an array are combined in some way to produce a cumulative result. In a prefix function the elements of an array are combined from left to right to produce the result, with each element of the result equal to the corresponding to the preceding element of the result combined with the corresponding element of the original array. Similarly, a suffix function determines the result from the cumulative operation of the elements that follow it, scanning through the array from right to left. The scan or prefix/suffix functions are often used in constructing parallel algorithms on graphs or other general data structures. They are also useful, because efficient implementations exist for parallel architectures. Typical operations include sum, product, maximum, minimum, logical AND, OR, PARITY.

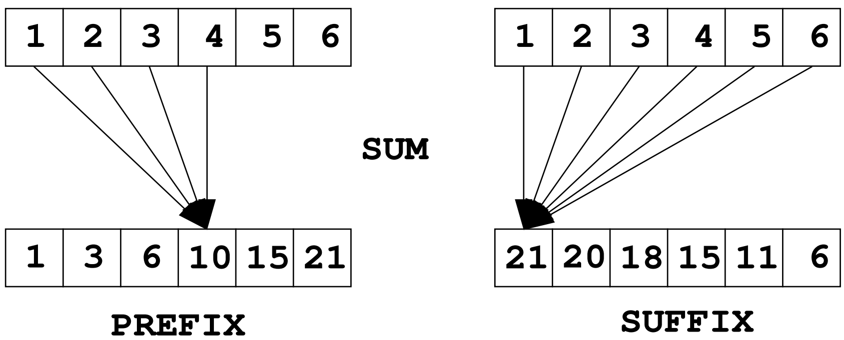


Figure 9: Prefix/Suffix SUM Operations

1.4.8 Generalised Communications

In certain applications (finite element codes for example) quite complicated data communication patterns can arise. In particular, data movement between arrays of different shapes and where elements are sent to the same location of the target array and combined in some way. Figure 10 illustrates such a situation

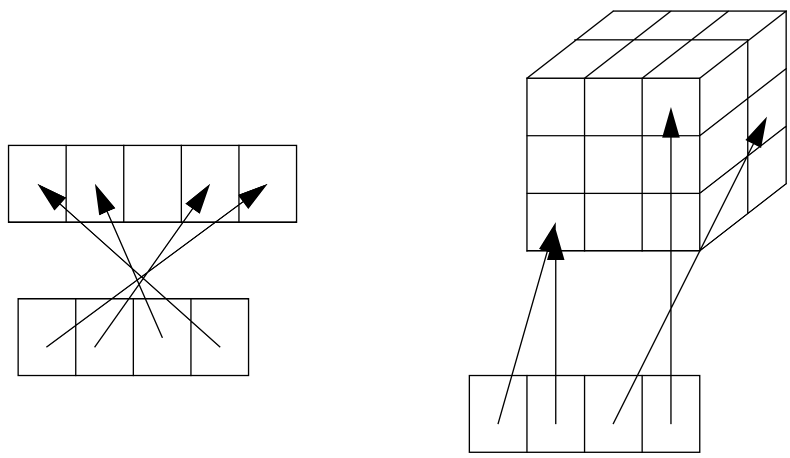


Figure 10: Examples of Generalised Communications

1.4.9 Building Blocks Summary

In the above sections we have only considered a set of basic building blocks for data parallel languages. However, they give a feel for the type of operations that are widely used in data parallel programming. Early languages only had a small number of basic facilities. Only recently has full functionality become available.

As we will see shortly, High Performance Fortran is based on Fortran 90. One reason for this being that Fortran 90 contains facilities for array operations. Such facilities have formed the core of data parallel languages to date. In a later section we give a brief overview of the array facilities in Fortran 90

1.5 Summary

In this chapter we have considered:

- parallel architectures,
- message passing and data parallel programming styles, and
- data parallel language building blocks

In the next chapter we shall consider the efforts of the High Performance Fortran Forum: the body responsible for the definition of HPF, before continuing with the actualities of HPF programming.

2 A Brief History of HPF

2.1 HPF History

The discussions on a High Performance Fortran standard were started at Supercomputing'91 where DEC had organised a discussion meeting for interested parties. This meeting led to the formation of the High Performance Fortran Forum (HPFF). The first meeting of the HPFF was held in Houston, Texas, in January 1992 and was attended by 130 people. Detailed work began in March when a working group of 40 people started fleshing out a standard with an intention to finish it by January 1993. A further eight meetings were held during 1992 leading to the publication of the HPF Specification v1.0 in May 1993.

The working group contained members from industry, universities and (US) government laboratories. A list of affiliations of those who attended more than two meetings is given in Table 1.

Table 1: Affiliations of participants of HPFF

Alliant Computer Systems Corporation	Meiko, Inc.
Amoco	nCUBE inc.
Applied Parallel Research	Ohio State University
Archipel	Oregon Graduate Institute
Convex	The Portland Group, Inc.
Cornell Theory Center	RIACS
Cray Research Inc.	Rice University
Digital Equipment Corporation	Schlumberger
Fujitsu	Shell
GMD	SUNY Buffalo
Hewlett Packard	Sun Pro, Sun Microsystems
IBM	Syracuse University
ICASE	Technical University, Delft
Intel Supercomputer	Thinking Machines
Lahey Computer	Unified Technologies
Lawrence Livermore National Laboratory	University of Stuttgart
Los Almos National Laboratory	University of Southampton
Louisiana State University	University of Vienna
MasPar Computer Corporation	Yale University

A wider discussion was facilitated by the creation of a mailing list for those interested in the work in progress. The HPFF has no official connection with recognised standards bodies. Work on Fortran standardisation is undertaken by the ANSI X3J3 committee which is effectively under direction from an international group WG5 (more formally ISO/IEC JTC1/SC22/WG5.) Some members of X3J3 were also active within HPFF and hence there is informal contact between the two groups.

2.2 HPFF Goals

HPFF goals were to define a language that offers:

- Data parallel programming (Single threaded, global name space, loosely synchronous parallel computation).
- Top performance with SIMD and MIMD machines with non-uniform memory access costs.
- Code tuning for various architectures.

These goals were addressed in the form of new intrinsics and first class language constructs.

Subsidiary goals were also established, including:

- portability of existing code
- efficient portability of new code
- maintained compatibility with existing standards (particularly Fortran 90),
- simplicity and ease of implementation
- open interface to other languages or programming styles.
- availability of compilers in the near future

2.3 The HPF Language

HPF is based upon the Fortran 90 programming language. Fortran 90 includes the FORTRAN 77 language and provides many new features including:

- Array operations
- Improved facilities for numeric computation
- User defined data types
- New control constructs
- Facilities for modular data and procedure definition
- New source form
- Optional procedure arguments and recursion
- Dynamic storage allocation
- Pointers

As noted previously, the Fortran 90 standard made an ideal basis for HPF. HPF extends the Fortran 90 language by defining new directives, language syntax and library routines. Some restrictions were necessary in order to accommodate the linear memory model inheritance of some Fortran 77 codes.

HPF adds the following *features* to those in Fortran 90.

- Support for controlling the alignment and distribution of data on a parallel machine.

- New data parallel constructs.
- An extended set of intrinsic functions and standard library providing much useful functionality at a high level of abstraction.
- **EXTRINSIC** procedures which standardise the interface with the other languages or styles
- Directives to address some sequence and storage association issues.

These features will be considered in detail in subsequent chapters.

2.4 Subset HPF

One of the goals of the HPFF was to promote the early adoption of HPF. As an aid to this a subset standard was defined in the hope that subset compilers could soon be produced. This is understandable since the effort involved in producing a full implementation is considerable, not least because HPF contains most of Fortran 90.

Many Fortran 90 features are included in the subset, for example nearly all of the array features and intrinsic functions. However, features not defined in Subset HPF include:

- free source form
- **POINTER** and **TARGET** attributes
- derived data types and generic operators
- **MODULE**s
- some intrinsic functions; character functions (**ADJUSTL**, **ADJUSTR**, **LEN_TRIM**, **REPEAT**, **SCAN**, **TRIM**, **VERIFY**), pointer association inquiry (**ASSOCIATED**), numeric inquiry, conversion functions (**ACHAR**, **ICHAR**), kind functions.
- new I/O features.

Similarly, there are some features in full HPF, but which were not included in subset HPF, again to promote early implementations of the language and because some of the more advanced features did not seem easily implementable. These include

- **REALIGN**, **REDISTRIBUTE** and **DYNAMIC** directives for dynamic data mapping
- the **PURE** function attribute for providing “safe” functions to be used in **FORALL** statements
- the multi-statement **FORALL** ... **END FORALL** construct
- the **HPF_LIBRARY** module
- **EXTRINSIC** functions
- The **INHERIT** directive used in transcriptive data mapping in procedures

Compilers for subset HPF are now available. The exercises in this course will use the HPF compiler from The Portland Group (PG) called pghpf. In the next section we outline the compilation process used by this compiler and describe how a simple program can be compiled and ran using the PG package.

2.5 An HPF Compiler

Although this discussion is specific to the Portland Group (PG) compiler, many of the points made are quite general and will apply to other HPF implementations. Only the barest details are included in this section. Greater detail is provided in documentation for the compiler.

2.5.1 The Portland Group Compiler: pghpf

The Portland Group HPF Compiler (**`pghpf`**) is a parallelising compiler for HPF. Basically it translates HPF/Fortran 90 codes into an equivalent FORTRAN 77 form with calls to message passing libraries. In addition, the Portland Group package contains automatic parallelisation capabilities, which we will not detail here. The compiler is designed to work on a variety of hosts (including Sun workstations, Cray T3D, Meiko CS2, Silicon Graphics workstations and the IBM SP2) and supports multiple communication protocols (PVM, PARMACS, MPI, and RPM (optimised PVM)), providing an automatic parallelisation of FORTRAN 77, Fortran 90 and HPF codes.

The **`pghpf`** compiler targets an SPMD (Single Program Multiple Data) programming model which is implemented by loading the same program image into each processor. The program running on each node makes calls to a Portland Group run-time library, which in turn makes calls to a standard message passing system, RPM in our case. Care is taken to address the unique communication characteristics of the target parallel system. The Portland Group runtime libraries take into account the communications to be performed and are optimised at two levels, details of which can be found in the user documentation.

The compilation process can be broken down into a number of stages:

- The HPF source text is preprocessed by **`pghpf`**.
- Syntax of the source text is checked.
- A FORTRAN 77 program with calls to Portland Group run-time libraries is generated.
- An executable is produced by compiling the resulting FORTRAN 77 code with **`f77`**, and linking to the appropriate Portland Group libraries.
- The program is then executed over a number of workstations, using RPM as the communications.

These steps are illustrated in Figure 11. The drivers for each of the stages are indicated in ellipses. The HPF source file is preprocessed by **`pghpf`**. The resulting program is processed in sequence by **`f77`** assembler and the linker. The executable code produced can then be loaded onto a number of processors.

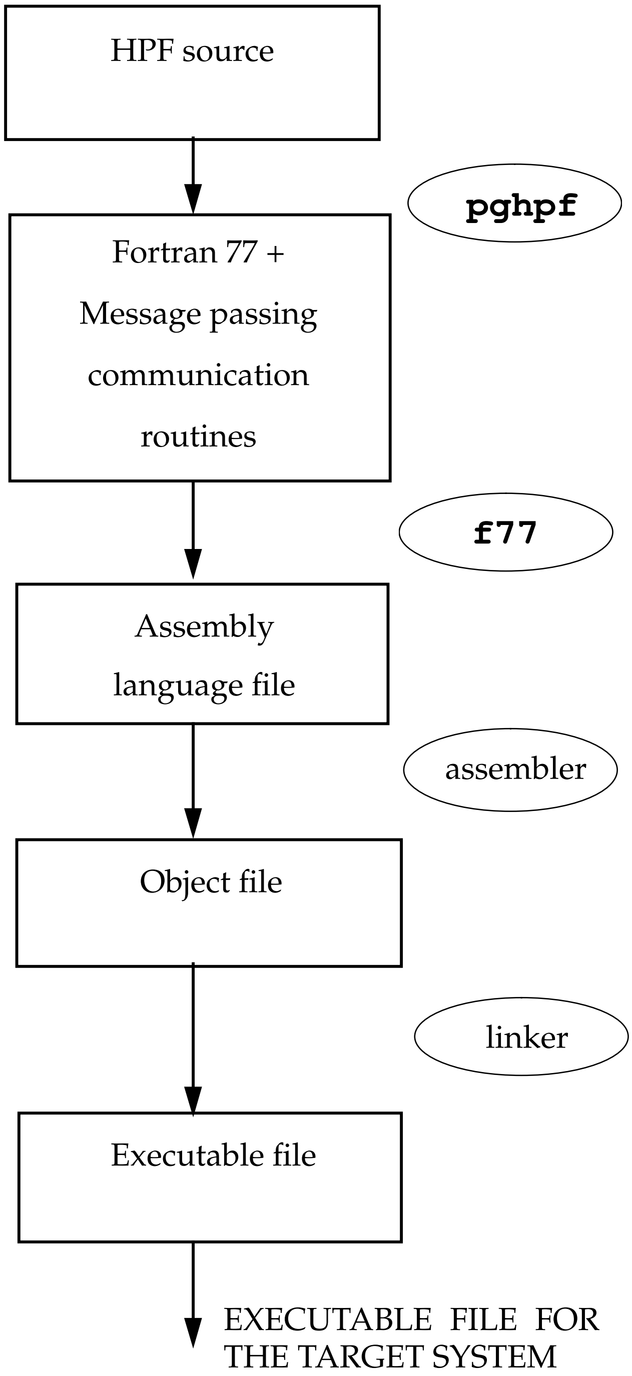


Figure 11: HPF compilation stages

2.5.2 Compiling and Running a Program

In all of the above stages a number of environment variables have to be set, depending on the configuration used, the target machines and communications protocol. Also at each stage various libraries have to be accessed and the files produced stored in particular places. To avoid unnecessary complications all the required commands have been collected in a script which executes each stage separately. The script is called **hpf** and has several command line options. The form of the command is

```
hpf [options] file
```

to execute the compilation procedure for a program with root name **file**. The command line options are as follows:

- **-help, -h**

  Print a help message, with the definitive description of how the script behaves

- **-compile**

  Performs translate stage and also the **f77** compilation of the resulting code to produce an SPMD executable, linking in Portland Group libraries

- **-run**

  Run SPMD program.

- **-nodes n**

  Run on **n** processors, where **n** is given as the next argument. The default is **n = 1**. If **n > 1** is specified, then **./MYHOSTS** file must exist (see later) and all stages of the compilation stage must include this option

- **-v**

  Verbose output

- **-profile**

  Turn on profiling, (needs to be included in all stages of the process)

  - for **-compile** it instruments f77 code with timing calls,

- with **-run** it produces an output detailing the cpu usage, memory usage and details of the message passing involved.

  -

Advanced Options

- **-cargs**

  Next argument is added at the start of the compilation arguments to xhpf

- **-largs**

  Next argument is added to the start of the link step argument list.

- **-largse**

  Next argument is added to the end of the link step argument list.

- **-VP**

  Give a verbose report on the parallelisation as it proceeds.

A typical sequence of calls to compile **file.hpf** into an executable named **file** which will run on a single Sun4 system, would be:

```
hpf -compile file
```

```
hpf −run file
```

2.5.3 Running on a Single Node

Consider the following simple example code, **hello.hpf** say.

```
PROGRAM hello
IMPLICIT NONE
INTEGER i, sum, a(10000)
a = 1
sum = 0
DO i = 1, 1000
   sum = sum + a(i)
END DO
PRINT *, `hello, the sum is ', sum
END PROGRAM hello
```

```
hpf −compile hello
```

Now run the program on a single node, by issuing the command:

```
hpf −run hello
```

This should result in:

```
hello, the sum is 10000
```

appearing on your terminal.

We are now interested in producing some timing information about the code. This is done with the **−profile** option on all three commands issued to run our simple example code. When profiling a code, it is important that all the input/output of the code is commented out, so as to get a better representation of the execution of the code itself. The first stage is done using the following line,

```
hpf −compile −profile hello
```

This includes the timing calls within the FORTRAN 77 code generated in this precompilation stage. The **−profile** option must be included in the running stage also,

```
hpf −run −profile hello
```

The output for this now includes the execution information, giving details of the cpu and memory usage and details of messages sent and received. The following is a typical output:

```
hello, the sum is        10000

cpu         real       user        says      ratio    node
   0*       0.07       0.02       0.01        45%   halite
total       0.07       0.02       0.01      0.45x

messages   send     send      send      recv     recv      recv
           cnt     total      avg       cnt     total      avg
   0*         0       0 b       0 b         0      0 b       0 b
total         0       0 b       0 b         0      0 b       0 b
```

The first set of information deals with the CPU related information. the first column lists the processor number (a * indicates which processor is printing the information). The next three columns show real (elapsed), user and system times, all in seconds. The fifth column shows the percentage of elapsed time the CPU was active (= **(user_time + sys_time)/real_time)**, and the last column indicates the processor name.

The message information indicates the number and amount of messages sent and received by each processor. For a single processor, clearly there are no messages.

This information can be used to tune codes by observing where the compute time is spent, how much communication was involved (and whether the data distribution was suitable for the application) and how the problem size should be scaled to achieve best performance out of the system.

2.5.4 Running on More than One Node

The **hpf** script can also be used to run the executable on many nodes. The compile and link stages are the same as with the single node execution.In order to tell the which hosts are being used, it is necessary to create a **MYHOSTS** file listing the machines to be used.

The training room has seven hosts,

- **young**,
- **sutherland**,
- **stewart**,
- **crawford**,
- **ramsay**,
- **blair** and
- **coventry**

If logged onto **young** in the training room and wanting to run a code on **sutherland**, **stewart** and **blair**, the **MYHOSTS** file could look like,

```
young
sutherland
```

```
stewart
blair
```

with the local machine top of the **MYHOSTS** list.The number of nodes required for any run of the code need only be specified at run time on the command line. The code is then run using the

```
hpf -run -nodes 4 <filename>
```

command.When changing the number of nodes at run time from 1 to more than 1, it is necessary to recompile the code with **-nodes** flag set. However, with this done, the executable can be run on any number of workstations, with the number specified at run time.

The profiling is done in the way as with a single processor, again the **-profile** option must be included at every stage of the process. However, now the information is given for each processor separately. In addition, min, max and average values for each quantity are given. The ratio listed in the last line of the CPU information table gives a measure of the speed-up factor achieved, calculated using the following **(user_time + sys_time)/real_time**.

This information can be used to tune codes by observing where the compute time is spent, how much communication was involved (and whether the data distribution was suitable for the application) and how the problem size should be scaled to achieve best performance out of the system.

2.6 Summary

In this chapter we considered the history, goals and current development status of HPF. The Portland Group HPF compiler, **pghpf**, has been introduced. We have considered the components of the compilation process and detailed the script provided to control the compilation and execution of HPF codes.

2.7 Exercise 1: Introduction to the Compiler

Write a Fortran 90 code to evaluate the discontinuous function:

$$0 \le x < 400: \quad y = x * x * x + C1$$

$$400 \le x < 1000: \quad y = x * x + C2$$

$$1000 \le x \le 10000: \quad y = x$$

- **Declarations**:

  Declare two vector integer arrays, **x** and **y**, each with 10000 elements, and two integer scalars, **C1** and **C2**.

- **Initialisation**:

  Initialise the vector **x** to be

```
x(i) = i, for all values of i from 1 to 10000
```

using the vector subscript array assignment of Fortran 90.

- **Calculation**:

  Assign the corresponding elements elements of y. Use either

  - the array subscript notation to assign array sections
  - or use the **`WHERE`** statments on the whole array ,

  Print out your answers for *y(50)*, *y(500)* and *y(1500)*. Check that your code gives the correct results (125005, 250010 and 1500 respectively).

- **Compilation**:

  Call your program **`ex1.hpf`**, and compile it using the script provided:

```
hpf -compile ex1
```

  This converts the code into FORTRAN 77 with calls to a communication library and then produces an executable.

- **Running on a single node**:

  Run the executable on a single processor, using the **`-run`** option for a single processor (check you answers agree with the ones above):

```
hpf -run ex1
```

- **Profiling:**

  Profile the code by including the **`-profile`** option on every command line. Remember to comment out all the input/output in the original code. Examine the output produced.

3 Fortran 90 Features

Fortran 90 is an international standard language, evolved from FORTRAN 77. In addition to all of FORTRAN 77, Fortran 90 contains a number of new features intended to make programs easier to write and with an obvious view to execution on parallel machines. These new features include,

- new source form (and improved syntax),
- new type declarations,
- array operations,
- derived data types and generic functions,
- dynamic memory allocation and pointers,
- powerful set of intrinsic functions,
- constructs to encourage modularisation of code,
- optional procedure arguments.

Of these features, the array processing features of Fortran 90 are of primary interest to us as a basis for HPF. There are two reasons for this. Firstly, array operations are provided as high level functions, with the hope that this will facilitate efficient implementation on vector and parallel machines. Secondly, algorithms based on array operations can be expressed concisely, thus increasing ease of programming, reducing the likelihood of error and allowing for concentration of effort on the application.

Since this course assumes a knowledge of Fortran 90, we briefly mention only some of key features of the language below. Further detail about Fortran 90 can be obtained from one of the many manuals and tutorials available on the subject.

3.1 New Source Form

Fortran 90 allows for a free source format, removing the restrictions FORTRAN 77 placed on the line layout. The main differences include; leading spaces are not significant, line continuation is specified with an ampersand character (&) at the end of a line, and comments can be placed anywhere following an exclamation mark (!). In addition the relational operators **.LT.**, **.LE.**, **.EQ.**, **.NE.**, **.GE.** and **.GT.** can be replaced by **<**, **<=**, **==**, **/=**, **>** and **=>** respectively. Also allowed are multiple statement lines, separated by a **;**, for example,

```
a = 0; b = a
```

Other improvments in syntax include variable names with over 31 characters, “_” in names and lower case. New statements are also introduced (often included in FORTRAN 77 compilers and taken form the MIL-SPEC standard) including the **END DO** statement to complete **DO** loops, **IMPLICIT NONE** and the **INCLUDE** statement.

3.2 New Type Declarations

Fortran 90 also supports a new form for declarations, including attributed type declarations. These are shown in the following with the equivalent FORTRAN 77 declarations.

FORTRAN 77	Fortran 90
`FUNCTION mandel(p)`	`FUNCTION mandel(p)`
`INTEGER N`	`IMPLICIT NONE`
`PARAMETER (N=1000)`	`INTEGER, PARAMETER :: N = 1000`
`REAL p(N,N), r(N,N)`	`REAL, DIMENSION(N,N) :: p, r`
`INTEGER count, mandel`	`INTEGER :: count = 10, mandel`
`DATA count /10/`	
`...`	

Note in particular, the use of **IMPLICIT NONE**, the **PARAMETER** attribute qualifying the **INTEGER** type declaration for **N**, the use of **::** as a separator for the declaration and a possible variable list, the data initialisation in the **INTEGER** statement (equivalent to a **DATA** statement with an implicit **SAVE**).

3.3 Array Expressions

Expressions may contain unindexed array variables, with the operations acting on whole arrays. However, this is restricted to operations between arrays which are conformable, that is, they must have the same rank and the same extents in each dimension (in addition, a scalar is conformable with any array). For example, consider the following FORTRAN 77 code and its equivalent Fortran 90 expression

```
   REAL A(100),B(100)
   DO 1, i=1,100
     A(i)=2.0
     B(i)=A(i)+SQRT(A(i))/2.0
1  END DO
```

```
REAL,DIMENSION(100)::A,B
A=2.0
B=A+SQRT(A)/2.0
```

3.4 Intrinsic Functions

Many of the intrinsic procedures contained in Fortran 90 can operate on arrays. The standard mathematical functions accept array arguments and operate elementally on each element of the array argument. Other useful functions are provided:

- array manipulations: **CSHIFT** and **EOSHIFT** for shifts along array axes, **TRANSPOSE** for the transpose of a matrix;
- reduction functions, including **SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ALL** and **ANY**;
- inquiry functions **SHAPE, SIZE, ALLOCATED, LBOUND** and **UBOUND**;
- array constructor functions **MERGE, SPREAD, RESHAPE, PACK** and **UNPACK** and
- other procedures, to do often used matrix multiplications (**MATMUL**), scalar products (**DOT_PRODUCT**) etc.

3.5 Subscript Triplets, Vector-Valued Subscripts

Subscript triplets facilitate the selection of a subset of array elements, which are often required in array assignment. These take the form *lower:upper:stride*, indicating the first element, last element and the jump between each element. For example the selection of interior points of an array is expressed in triplet notation as:

FORTRAN 77	Fortran 90
`REAL A(10,10),B(10,10)`	
`DO 1, j=1,8,2`	
`DO 2, i=2,9`	`REAL, DIMENSION(10,10) :: A,B`
`A(i,j) = B(i+1,j+1)`	`A(2:9,1:8:2) = B(3:10,2:9:2)`
`2 CONTINUE`	
`1 CONTINUE`	

Array assignments also make use of vector subscripts,

FORTRAN 77	Fortran 90
`INTEGER n`	
`PARAMETER (n=100)`	`INTEGER, PARAMETER :: n=100`
`REAL X(n)`	`REAL, DIMENSION(n) :: X`
`DO 1, i=1,n`	
`X(i) = i/100.`	`X = (/ (i, i=1,n) /) / 100.`
`1 CONTINUE`	

3.6 The WHERE Control Construct

Operations can be restricted to a set of array elements with the **WHERE** construct, which allows for a parallel **IF** statement. Again we show a sample FORTRAN 77 code and then the equivalent Fortran 90.

FORTRAN 77	Fortran 90
`REAL A(1024)`	
`DO 1, i=1,1024`	`REAL, DIMENSION(1024) :: A`
`IF(A(i).GT.0.0) THEN`	`WHERE ( A > 0.0 )`
`A(i)=1.0/A(i)`	`A = 1.0 / A`
`ELSE`	`ELSE WHERE`
`A(i)=0.0`	`A = 0.0`
`END IF`	`END WHERE`
`1 CONTINUE`	

3.7 Interface Blocks

Fortran 90 allows a way to specify an explicit interface for an external procedure. This gives information on the name of the procedure, types of passed and returned parameters and whether an argument may be changed in the procedure. In effect, the interface block is simply a copy of the procedure without the executable statements. Use of explicit interface definitions allows incorrectly formed calls to be detected at compile time.

For example,

```
INTERFACE
  REAL FUNCTION DISTANCE(A,B)
    REAL, INTENT(IN) :: A, B
  END FUNCTION DISTANCE
END INTERFACE
```

describes the interface for a function of two real dummy arguments, **`A`** and **`B`**, which are not modified by the function, and which returns a real, **`DISTANCE`**.

Note also the introduction of the **`INTENT`** attribute, used to qualify the declaration and tell the compiler that these variables will not be altered during the procedure. This allows the compiler to generate more efficient code by removing unnecessary copying.

3.8 Derived Data Types and Generic Functions

Fortran 90 allows the user to define new data types, created from a collection of the intrinsic types. These are similar to the concept of structures or records in other languages. For example, consider a derived data type to describe a coordinate point, containing an x and y value. A typical derived type for this example might be,

```
TYPE POINT
  INTEGER :: X, Y
END TYPE
```

with this derived data type defined, we can declare an object of this type, and then assign values to the separate components,

```
TYPE(POINT) :: START, FINISH
START%X = 0 ; START%Y = 0
FINISH%X = 12; FINISH%Y = 6
```

These objects can also be combined. Although the meaning of **`START%X - FINISH%X`** is clear (subtraction of two integers), what is not clear is the meaning of **`START-FINISH`**. Fortran 90 allows for an unambiguous definition of this operation, by overloading an existing operator. This allows the programmer to use the same name to describe a number of functions which describe a generic operation. This overloading is implemented by using an **`INTERFACE`** block, for example,

```
INTERFACE OPERATOR(-)
   FUNCTION DIFF(A,B)
     TYPE(POINT) :: DIFF, A, B
   END FUNCTION
END INTERFACE
```

for the module defining the subtraction operation on two objects of the **POINT** type,

```
FUNCTION DIFF(A,B)
   TYPE(POINT) :: DIFF, A, B
   DIFF%X = A%X - B%X
   DIFF%Y = A%Y - B%Y
END FUNCTION DIFF
```

So, for example, the operation **START-FINISH** would now be defined to be the subtraction of the individual elements of the two derived types.

3.9 Modules

Fortran 90 encourages the writing of modular code by the introduction of Modules. Modules can be used to collect data and/or sub-programs. The inclusion of modules into Fortran 90 is intended to improve reliability (modules can be developed and tested in isolation and included into a program with the **USE** statement) and to allow re-use of code (access to data/sub programs in a controlled way).

For example,

```
MODULE POINT_MODULE
   TYPE POINT
      REAL :: X, Y
   END TYPE
   INTERFACE OPERATOR(-)
      MODULE PROCEDURE DIFF
   END INTERFACE
...
CONTAINS
   FUNCTION DIFF(A,B)
      TYPE(point) :: DIFF, A, B
      DIFF%X = A%X - B%X
      DIFF%Y = A%Y - B%Y
   END FUNCTION DIFF
   ...
END MODULE POINT_MODULE
```

Here we have collected the data structures and procedures from the previous subsections, the first part of the module containing the specification statements (derived data type and interface block for the generic function), the second part of the module (after the **CONTAINS** statement) list the subprograms contained within the module.

This module (and all the data objects/ sub programs contained) is accessed by the following statement,

```
USE POINT_MODULE
```

3.10 Dynamic Memory Allocation and Pointers

Fortran 90 allows for dynamic memory allocation by use of the **ALLOCATE** statement. In this way an allocatable array can be declared using the **ALLOCATABLE** attribute as follows,

```
REAL, DIMENSION(:,:), ALLOCATABLE :: a
```

specifying the array name and rank, but leaving the array bounds undefined until later, for example when the array size is read in,

```
INTEGER :: n
READ(*,*) n
ALLOCATE( a(0:n,0:n) )
...
```

When the array is no longer needed, its memory allocation can be freed using

```
DEALLOCATE( a )
```

Pointers allow the programmer to refer to different objects during the execution of a program. Consider the following example, using pointers **P1** and **P2** (specified by attribute **POINTER**) to refer to integers A and **B** (with attribute **TARGET**),

```
INTEGER, TARGET :: A, B
INTEGER, POINTER :: P1, P2
A = 1; B = 2      ! variable assignment
P2 => B; P1 => P2 ! pointer assignment (association)
PRINT *, A, B, P1, P2! will print: 1 2 2 2
P1 => A; P1 = P2  ! targets used in assignment
PRINT *, A, B, P1, P2! will print: 2 2 2 2
```

in the **P1 = P2** statement, the assignment is made between the target variables (**A** and **B**) themselves.

3.11 New Control Constructs

The **SELECT CASE** construct is included in Fortran 90 to replace the computed **GOTO** in FORTRAN 77 as a means of selecting one of several options for execution. For example,

```
INTEGER :: A
...
SELECT CASE(A)
CASE(1)
     CALL SUNDAY
CASE(2:6)
   CALL WEEKDAY
CASE(7)
   CALL SATURDAY
CASE DEFAULT
   CALL BADDAY
END SELECT
```

In the above **SELECT CASE** construct, only one expression is evaluated for testing (the value of integer **A**) and this evaluated expression can only belong to one a of a set of values (with **CASE DEFAULT** covering erroneous values not covered in the other **CASE**s).

3.12 Dummy Argument Declarations

Fortran 90 extends the possibilities for the declaration of dummy arguments in procedures. FORTRAN 77 allowed arrays passed to be declared with explicit shapes (both static and adjustable extents) in which the extents of the dummy argument were explicitly given, though the extents could be specified as procedure arguments. For example,

```
SUBROUTINE s1(a,b,c,n,m)
INTEGER n,m
REAL a(100,100)          ! static
REAL b(n,m)              ! adjustable
REAL c(-10:20,n:m)       ! adjustable
...
```

Also contained in FORTRAN 77 is the concept of an assumed size array. An assumed size declaration allows the programmer to specify the extent of an array (but only for the last extent, the others must be explicitly declared).

```
SUBROUTINE flops(a,b,n)
REAL a(*)
REAL b(0:n,0:n,*)
```

However, this feature relies on the assumption of a linear memory model (a large problem in distributed memory machine). As such, assumed size arrays are on the depricated list of features, which outlines the features that become redundant on Fortran 90, and which could be dropped from future versions of the language.

New to Fortran 90 are assumed shape dummy argument declarations. In this case, shape of the array is not specified (the rank of the array is specified but the extents are not listed). The declaration of each dimension is of the form ***[lower-bound]*:**. So for example,

```
SUBROUTINE s2(a,b,n)
IMLICIT NONE
INTEGER :: n
REAL a(:,:,:)           ! 3-D array
REAL b(0:,n:)           ! 2-D array, starting at (0,n)
```

Also related to this are automatic arrays. These are typically used in situations where a procedure may need a local array who size varies depending on the input to the procedure, for example in a subroutine which uses a work array. For example,

```
PROGRAM any_size
IMPLICIT NONE
INTEGER :: n, m
READ *,n,m
CALL sim(n,m)
END

SUBROUTINE sim(n,m)
IMPLICIT NONE
INTEGER :: m
REAL a(n,m), b(m)       ! local arrays,
...                     ! not passed as arguments
END
```

3.13 Keywords and Optional Arguments

Fortran 90 allows the programmer to declare procedure arguments with the **OPTIONAL** argument, allowing them to be omitted from the procedure call. If the optional arguments are included in the procedure call they are “mapped” to the dummy arguments either by position (order in the argument list) or by use of keywords, which allows the programmer to explicitly indicate which argument is being provided. Whether an argument has been included or not can be tested for, using the **PRESENT** function. For example,

```
CALL set_corners(a)
CALL set_corners(a,value=2)
END
```

```
SUBROUTINE set_corners(a,value)
IMPLICIT NONE
INTEGER, INTENT(IN), OPTIONAL :: value
INTEGER, DIMENSION(:,:) :: a
IF (PRESENT(value)) THEN
       a(::SIZE(a,dim=1)-1,::SIZE(a,dim=2)-1) = value
ELSE
       a(::SIZE(a,DIM=1)-1,::SIZE(a,DIM=2)-1) = 0
END IF
END
```

Procedures with optional arguments also require explicit interface blocks to be given.

3.14 Intrinsic Procedures

Fortran 90 provides a large set of intrinsic functions to provide a range of widely used operations. There are five main classes: elemental procedures (specified for scalar arguments, but also for array arguments acting element by element), inquiry functions (properties of arguments), transformational function, nonelemental subroutines and the array intrinsic procedures. In each case the intrinsic procedures are listed below, though not in great detail.

3.14.1 Elemental Procedures

```
CEILING(A), FLOOR(A), MODULO(A)

ACHAR(I), ADJUSTL(STRING), ADJUSTR(STRING),
CHAR(I[,KIND]), IACHAR(C),
ICHAR(C),INDEX(STRING,SUBSTRING[,BACK]), LEN(STRING),
LEN_TRIM(STRING), LGE(STRING_A,STRING_B),
LGT(STRING_A,STRING_B), LLE(STRING_A,STRING_B),
LLT(STRING_A,STRING_B), REPEAT(STRING,NCOPIES),
SCAN(STRING,SET[,BACK]), TRIM(STRING),
VERIFY(STRING,SET[,BACK])

BIT_SIZE(I), BTEST(I,POS), IAND(I,J), IBCLR(I,POS),
IBITS(I,POS,LEN), IBSET(I,POS), IEOR(I,J), IOR(I,J),
ISHFT(I,SHIFT), ISHFTC(I,SHIFT[,SIZE]), NOT(I)

KIND(X), SELECTED_INT_KIND(R),SELECTED_REAL_KIND(P,R)

DIGITS(X), EPSILON(X), HUGE(X), MAXEXPONENT(X),
MINEXPONENT(X), PRECISION(X), RADIX(X), RANGE(X), TINY(X)

EXPONENT(X), FRACTION(X), NEAREST(X,S), RRSPACING(X),
SCALE(X,I), SET_EXPONENT(X,I), SPACING(X)

LOGICAL(L[,KIND])

CALL MVBITS(FROM,LEN,TO,TOPS)
```

3.14.2 Inquiry Functions

```
ALLOCATED(ARRAY), LBOUND(ARRAY[,DIM]), SHAPE(SOURCE),
```

```
SIZE(ARRAY[,DIM]), UBOUND(ARRAY[,DIM])
ASSOCIATED(POINTER[,TARGET]), PRESENT(A)
DIGIT(X), EPSILON(X), HUGE(X), MAXEXPONENT(X),
MINEXPONENT(X), PRECISION(X), RADIX(X), RANGE(X), TINY(X)
```

3.14.3 Transformational Functions

```
REPEAT(STRING,NCOPIES), TRANSFER(SOURCE,MOLD[,SIZE]),
TRIM(STRING)
```

3.14.4 Nonelemental Subroutines

```
DATE_AND_TIME([DATE][,TIME][,ZONE][,VALUES]),
MVBITS(FROM,FROMPOS,LEN,TO,TOPOS), RANDOM_NUMBER(HARVEST),
RANDOM_SEED([SIZE][,PUT][,GET]),
SYSTEM_CLOCK([COUNT][,COUNT_RATE][,COUNT_MAX])
```

3.14.5 Array Intrinsic Procedures

```
ALL(MASK[,DIM]), ANY(MASK[,DIM]), COUNT(MASK[,DIM]),
MAXVAL(ARRAY[,DIM][,MASK]), MINVAL(ARRAY[,DIM][,MASK]),
PRODUCT(ARRAY[,DIM][,MASK]), SUM(ARRAY[,DIM][,MASK]),
MERGE(TSOURCE,FSOURCE,MASK), PACK(ARRAY,MASK[,VECTOR]),
SPREAD(SOURCE,DIM,NCOPIES), UNPACK(VECTOR,MASK,FIELD)
RESHAPE(SOURCE,SHAPE[,PAD][,ORDER])
MAXLOC(ARRAY[,MASK]), MINLOC(ARRAY[,MASK])
CSHIFT(ARRAY,SHIFT[,DIM]),
EOSHIFT(ARRAY,SHIFT[,BOUNDARY][,DIM]), TRANSPOSE(MATRIX)
DOT_PRODUCT(VECTOR_A,VECTOR_B), MATMUL(MATRIX_A,MATRIX_B)
```

3.15 Summary

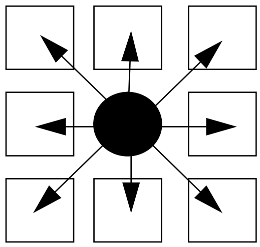
Clearly Fortran 90 provides an excellent starting place for any data parallel programming language. Indeed, in some cases the facilities provided in Fortran 90 may be all that is required to implement a data parallel algorithm. However, HPF goes further in not only providing array based operations but more importantly by providing the directives to distribute and align the data.

3.16 Exercise 2: The Game of Life

The aim of this exercise is to show how Fortran 90 can be used to program the Game of Life, a simple grid based problem with complex behaviour. It will show how Fortran 90 can be used to produce code in a very neat form and exposes the potential for coding in a data parallel programming style.

3.16.1 The Game of Life

The game of life is a simple cellular automaton where the world is a 2D grid of cells which have two states: alive or dead. At each iteration the new state of a cell is determined by the state of its neighbours at the previous iteration. This includes both the nearest neighbours and diagonal neighbours, i.e.



The rules for the evolution of the system are;

- if a cell has exactly two alive neighbours it maintains state.
- if it has exactly three alive neighbours it is alive.
- otherwise, it is dead.

Your code will need to

1. Initialise board

   Start loop

2. Print board
3. Calculate number of neighbours
4. if (neighbours = 3) then live
   if (neighbours < 2) or (neighbours > 3) then die

   End loop

The number of neighbours can be calculated using shifts, e.g.,

```
target = CSHIFT(source, shift, dimension)
```

sets **`target`** to be the same as **`source`** but with its elements shifted a distance **`shift`** along dimension of the array **`dimension`**. For example,

```
target = CSHIFT(source, -1, 1)
```

would set

```
target(i) = source(i - 1)
```

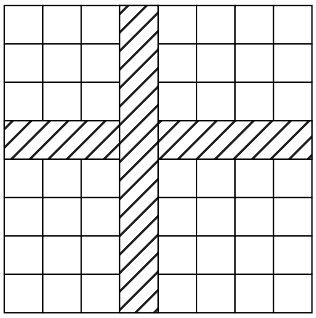
`CSHIFT` automatically performs periodic boundary conditions. Otherwise references would be made to elements outside the bounds of the array.

3.16.2 The Exercise

A template for this exercise is provided, which includes pointers to the completion of this exercise and also all the print statements necessary for viewing the results.

- **Initialisation**:

  Use array syntax to initialise a board (an **N×N INTEGER** array with value 1 for a live cell and 0 for a dead one) with the following pattern, where shaded cells are alive, where **N = 8** in this case. Set the row and column nearest the centre (i.e. **N/2**)



- **Print board:**

  The board can be printed either as a plain text file using the standard Fortran print format statements, or using the following pieces of code can be included to produce a series of bitmaps which can be animated using **xv**.

```
!      Include in the declarations; strings for filenaming
       CHARACTER (LEN=10) :: picfile
!...
! Include in the time loop
! hack to write time into a string
       WRITE(picfile,fmt='(''life'',I2.2,''.pgm'')') count

       OPEN(UNIT=10,FILE=picfile)
! write the board to a pgm file for viewing
       WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 1
! must be a capital P for the MAGIC NUMBER
       WRITE(10,*) board
        CLOSE(10)
```

  This should, when run, produce **life_*.pgm** files which can be viewed using **xv**. Alternativly, these files can be viewed as animation using,

```
xv -expand 10 -wait 0.5 -wloop -raw *.pgm
```

- **Update**:

Declare an array the same size as the board (**N**×**N**)to contain the number of neighbours for each point. Include all nearest neighbours, including diagonal neighbours. This update can be done using the following Fortran 90 features.

- Use **CSHIFT** to calculate the number of neighbours. In order to access the diagonal neighbours you may need to used nested **CSHIFT**s. For example,

```
target = CSHIFT(CSHIFT(source, 1, 2), -1, 1)
```

  This would shift array **source** initially in the 2nd dimension and then in the 1st.
- Use **WHERE** to decide whether to create or kill new organisms at each grid point.

- **Compilation**:

  Compile your code (**life.hpf**, say, board size **N=8** for the test, for 10 iterations of evolution) with the script as in the previous exercise (**hpf -compile life**) to produce an executable **life**.
- **Single node**:

  Run the code on a single processor for 10 iterations to check produce a set of (**hpf -run life**).

  Use **xv** to view the resulting arrays to check your code is working.

3.16.3 Extra Exercise 1

CSHIFT was used to count the number of neighbouring live cells. You can optimise the code to do the count with only four **CSHIFT** operations instead of eight. This requires the use of a temporary array the same size as **board**. For each array element,

- create the temporary by adding the current value of **board** to the value of the left and right partners (two **CSHIFTS**)
- find the upper and lower partners for this temporary array (two more **CSHIFTS**) and add these to the temporary array
- each element now contains the number of neighbours, plus the original value of the cell. Subtract the original state to obtain the number of neighbours with four **CSHIFT**s.

3.16.4 Extra Exercise 2

Try to count and display the number of alive/dead/new/killed cells at each iteration. You will probably need to use some additional Fortran 90 features.

3.16.5 Extra Exercise 3

Define an extra array which is used for display. In this array keep track of which generation (modulo 256) that the cell was born. This array can be used for a colour display and so the age of the cell can be seen. To print out this colour array, change the header statement from

```
WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 1
```

to

```
WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 255
```

4 Data Mapping 1

4.1 Data Distribution

One of the main ideas behind the data parallel paradigm is processing and manipulating large arrays. Expressing the operations as array operations opens up the possibility of executing similar operations on different array elements simultaneously (*ie*, in parallel).

On a distributed memory machine each processor has some local memory on which it can act directly, and the large data arrays must be distributed across many different processors. In the message passing style, this distribution of data across the processors is done explicitly by the programmer. However, data parallel languages typically contains compiler directives allowing some control over the data layout.

How the data is shared between processors depends on the application and how the array will be used. HPF offers a number of different ways that this can be done. The main thing to keep in mind is that we want to choose a distribution which maximises the ratio of local work to communication, thus maintaining an efficient parallel code.

4.2 HPF Compiler Directives

As this is the first time we have met HPF compiler directives, a short aside is necessary on the syntax of HPF compiler directive, as formally outlined in the HPF specification. These are as follows:

hpf-directive-line	is	*directive-origin hpf-directive*
directive-origin	is	**`!HPF$`**
	or	**`CHPF$`**
	or	**`*HPF$`**
hpf-directive	is	*specification-directive*
	or	*executable-directive*
specification-directive	is	*processors-directive*
	or	*align-directive*
	or	*distribute-directive*
	or	*dynamic-directive*
	or	*inherit-directive*
	or	*template-directive*
	or	*combined-directive*

	or	*sequence-directive*
executable-directive	is	*realign-directive*
	or	*redistribute-directive*
	or	*independent-directive*

The *directive-origin* typically starts in the first column of a line of code, and tells the compiler that some additional information is going to be provided. By starting these statements with `!`, `C` or *, the directive is effectively a comment to any other Fortran compiler, thus adding to the portability to serial machines of codes written in HPF. Also it reinforces the idea that these directives are only advisory in nature, and the compiler is allowed to ignore them as it sees fit.

In the following notes we will outline the meaning and uses of the various directives.

4.3 The DISTRIBUTE Directive

This is the main directive for specifying the distribution of arrays, providing a distribution for each dimension of the array. The format of this directive is as follows:

```
!HPF$ DISTRIBUTE a(distribution)
!HPF$ DISTRIBUTE (distribution) :: a, b
```

where the ***distribution*** is a comma separated list of the distribution for each array element. The second form of this directive is a combined directive, allowing for Fortran 90 like declaration lists.

The possibilities for the distributions available are listed below.

4.3.1 Block Distribution

One of the most common forms of distribution is known as a block distribution. As illustrated by the example in Figure 12, this means that each processor contains a single contiguous block of the array.

```
 REAL, DIMENSION(16) :: a
!HPF$DISTRIBUTE (BLOCK) :: a

 P1  P2  P3  P4
 1   5   9   13
 2   6   10  14
 3   7   11  15
 4   8   12  16
```

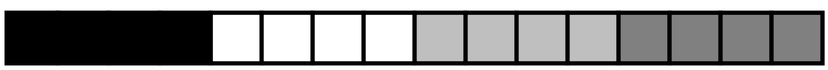


Figure 12: Example of a block distribution

The size of each processor's block is equal to the number of elements in the array divided by the number of processors. This type of distribution is used for regular domain decomposition in fields like computational fluid dynamics and QCD, where most operations on an array element only involve its nearest neighbours. It ensures that neighbouring elements are normally on the same processor and therefore minimises costly inter-processor communication.

4.3.2 Cyclic Distribution

Another regularly used distribution is the cyclic distribution. The first element is on the first processor, the second on processor 2 and so on; this is similar to the way that a pack of cards is normally dealt.

```
      REAL, DIMENSION(16) :: a
!HPF$ DISTRIBUTE (CYCLIC) :: a
```

P1	P2	P3	P4
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure 13: Example of a cyclic distribution

As shown in Figure 13, if a 16 element array is distributed across 4 processors, processor 1 has elements 1, 5, 9 and 13. This type of distribution can be regarded as similar to Scattered Spatial Decomposition, effectively dividing the data (and hence work) randomly (assuming the problem contains no periodicity). It is useful for applications where the work required for each element varies significantly throughout the array and locality is not important. Examples include ray-tracing and edge detection.

4.3.3 Degenerate Distribution

The final common distribution is degenerate or serial. The degenerate dimensions of an array are not distributed and all elements in that dimension are associated with the same processor.

```
     REAL, DIMENSION(8,4) :: a
!HPF$DISTRIBUTE (CYCLIC,*) :: a
```

P1	P2	P3	P4
1,1 5,1	2,1 6,1	3,1 7,1	4,1 8,1
1,2 5,2	2,2 6,2	3,2 7,2	4,2 8,2
1,3 5,3	2,3 6,3	3,3 7,3	4,3 8,3

1,4 5,4   2,4 6,4   3,4 7,4   4,4 8,4

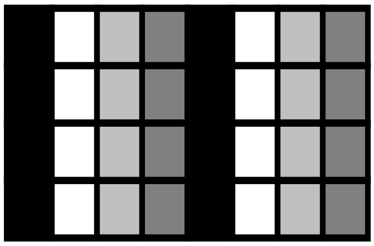


Figure 14: Example of a degenerate dimension

This type of distribution is useful for a dimension which is an array of attributes associated with the same point on a grid, e.g., the temperature, pressure and density at a particular point in space. All the information associated with the same grid point is stored on the same processor.

4.3.4 More General Distributions

In addition to the regular **`BLOCK`**, **`CYCLIC`** and degenerate distributions, HPF also provides other possibilities in the form of **`BLOCK(n)`**, and **`CYCLIC(n)`**. In each case the array is split into contiguous chunks of size n, and these blocks are distributed with the particular distribution.

For example, **`CYCLIC(2)`**, would collect chunks with 2 elements and distribute these cyclically:

```
REAL, DIMENSION(16) :: a
!HPF$ DISTRIBUTE (CYCLIC(2)) :: a
```

Figure 15: Example of a CYCLIC(2) distribution

Similarly **`BLOCK(n)`** distributes chunks of size **`n`**, except in this case, there would only be one chunk per processor. This means there must be enough processors to take all the chunks. For example,

```
REAL, DIMENSION(16) :: a
!HPF$  DISTRIBUTE (BLOCK(4)) :: a
```

would be ok as the number of processors >= (16 elements)/(chunks of size 4). Similarly **`BLOCK(5)`** would still be conforming, although now some processors would have less data (and hence work) than others. However, **`BLOCK(3)`** would be non-conforming.

4.4 Summary

HPF supports a method of data distribution which should be sufficiently flexible for many requirements. Distributed data can be operated on in a natural way using array syntax or using the other data parallel features provided by HPF and discussed in the next section.

4.5 Exercise 3: Life Distributed

In this exercise we return to Exercise 2 and rewrite the codes including explicit data distribution directives.

- **Distribute Data**:

  Introduce data mapping directives into the code to distribute the data. So, for example, for 2D arrays **source** and **target** you could use

```
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: source, target
```

  to distribute the elements of these arrays in a block form.

- **Compilation**:

  Compile the codes as before, using the **hpf** script. Increase the size of your array to **N=100**.

- **Single Node**:

  Run the executable on a single node. Check it gives the same results as before.

- **Multiple Nodes**:

  First create a **MYHOSTS** file in the working directory with four workstation names including your own (the training room machines are called **young**, **sutherland**, **stewart**, **crawford**, **ramsay**, **blair** and **coventry**). You need to have a **MYHOSTS** file in each directory where you run executables on more than one processor).

  Run the code on four processors, using

```
hpf -run -nodes 4 life
```

  and check you still get the same answer!

- **Profiling**: Repeat the last stages with the **-profile** option set in each command. Remember to comment out all the input/output statements when profiling.

  Use the resulting information to investigate how the performance of these examples changes with different data distributions. In particular profile the codes and see how the amount of communication changes. What is the most efficient distribution for the Game of Life and why?

  For example, change the distribution of the 2D arrays in the Game of Life from **(BLOCK,BLOCK)** to, say, **(CYCLIC,CYCLIC)** or using a degenerate distribution.

5 HPF Parallel Features

5.1 Introduction

The aim of HPF is to provide a set of language constructs which allow the simple expression of operations on data which can be applied in parallel should there be a parallel resource available. This desire to exploit the data parallel programming model is the driving force behind HPF, and as such features widely in this course.

In this chapter we will consider data parallel features that are new in HPF and which express operations to be performed on multiple elements of arrays. As a basis for this, HPF uses the array operations from Fortran 90, such as array assignment, masked assignment, array sections, shift operations, the **WHERE** statement and generalises these by allowing more array operations, expressed in a simpler way. The main expression of data parallelism provided by HPF is via the **FORALL** statement, the **INDEPENDENT** directive and through the use of new intrinsic functions. The **PURE** attribute is also provided to increase the generality of these other statements.

To begin, we look at two examples of code segments which respectively can and cannot be parallelised using simple methods.

Example: stencil function: simple parallelisation of the loop is possible,

```
DO i= 1, N
   A(i) = B(i-1) + B(i) + B(i+1)
END DO
```

as each operation is independent of the last

Example: Fibonacci series: No simple parallelisation of the loop is possible:

```
F = 1
DO i = 3, N
   F(i) = F(i-1) + F(i-2)
END DO
```

as each iteration depends on the result of the last.

We begin the discussion of the data parallel features new to HPF with the **FORALL** statement, the main first class language construct to be added to Fortran 90.

5.2 The FORALL Statement

The **`FORALL`** statement is a data parallel construct which defines the assignment of multiple elements in an array but without enforcing an order on the assignments to individual elements. It can be thought of as a generalisation of Fortran 90 array assignment, though focusing on the more fine-grain parallelism of the problem.

Typically, the FORALL statement is useful for assignments based on array index,

```
1)     FORALL (I=1:N, J=1:N) A(I,J) = 1.0 / REAL(I+J)
```

expression of irregular data motion

```
2)     FORALL (I=1:N) B(I) = A(I,I)
```

and whole array operations with user defined functions.

```
3)     FORALL (I=1:N, J=1:N) F(I,J) = FORCE( R(I,J), M(I,J))
```

The reason behind providing **`FORALL`** is to create a parallel construct which guarantees identical results if applied in serial or in parallel. This does not mean that **`FORALL`** is a general purpose parallel loop: the use of **`FORALL`** is restricted to assignments and the statement is executed in four well defined stages (similar to the execution of an array assignment). Compared with a simple **`DO`** loop, which executes on each data element in a well defined order, the order of execution on the data elements in a **`FORALL`** is undefined by HPF and is implementation dependent. In this sense, it is not correct to think of a **`FORALL`** statement as describing an iterative process, instead comparison with array assignments is more useful.

There are two forms of **`FORALL`** in HPF. The **`FORALL`** statement, containing a single assignment and the **`FORALL`** construct, which contains multiple assignment statements, but can always be written as a series of **`FORALL`** statements.

The main point to note is that the **`FORALL`** construct is NOT supported in subset HPF.

5.2.1 FORALL Syntax

The syntax for the **`FORALL`** statement is as follows:

forall-stmt	is	**`FORALL`** *forall-header forall-assignment*
forall-header	is	(*forall-triplet-spec-list* [, *scalar-mask-expr*])
forall-triplet-spec	is	*index-name* = *subscript* : *subscript* [: *stride*]
forall-assignment	is	*assignment-statement*
	or	*pointer-assignment-statement*

There are however, some restrictions on this:

1. in the *forall-triplet-spec*, the *index name* must be a scalar variable, and neither the *subscript* nor *stride* of a *forall-triplet-spec* can contain a reference to another varia-

ble in the *forall-triplet-spec-list*. e.g.

```
FORALL(I = 1:N, J = 1:I) A(I,J) = A(I,J) / A(I,I)
! NON-CONFORMING as value of J depends on value of I
!Work around using a mask, as follows
FORALL(I = 1:N, J = 1:N, J .LT. I) A(I,J) = A(I,J) / A(I,I)
```

2. any procedure used in either the *scalar-mask-expr* or the *forall-assignment* must be **PURE**. We will meet **PURE** procedures later, but the main idea behind them is that they have no side effects, so can safely be used in parallel execution.
3. the *forall-assignment* must not make multiple assignments to the same element, as the result depends on the order or execution and so is not defined for parallel execution. e.g. the following is undefined in HPF,

```
FORALL(I = 1:N) S = X(I)   ! non-conforming
```

Similarly, for the **FORALL** construct the syntax is:

forall-construct	is	**FORALL** *forall-header*
		forall-body-stmt
		[*forall-body-stmt*]
		END FORALL
forall-body-stmt	is	*forall-assignment*
	or	*where-stmt*
	or	*where-construct*
	or	*forall-stmt*
	or	*forall-construct*

The additional constraints on the **FORALL** construct are:

1. any procedure used in the *forall-body-stmt* must be **PURE**.
2. for a *forall-stmt* or a *forall-construct* nested within a *forall-construct*, the inner **FORALL** must not redefine any *index-name* in the outer *forall-construct*.

The **FORALL** construct is not in subset HPF.

5.2.2 Interpretation of a FORALL

The **FORALL** statement, which can be a single statement as above or a multi-statement construct, defines operations on a range of index values where an optional mask expression is true. This operation is executed in four stages, which we outline below, referring to the **FORALL** in the following example at each stage:

```
A(1) = 2; A(2) = 3; A(4) = 4
FORALL(I=1:6, A(I).NE.0) B(I) = 1.0 / A(I)
```

1. **Compute the valid set of index values**: This is defined in the first part of the **`FORALL`** statement in *forall-triplet-spec-list*. This can be done in any order. In our example the **`FORALL`** index range is values from **`1`** to **`6`**.
2. **Define the active set of index values**. This corresponds to the subset of the valid index values for which the (optional) *scalar-mask-expr* evaluates to **`.TRUE.`**. These are the actual index values for which the assignment will be made, and the computation of them could be done in any order. In this example the active set are all those indices **`1, 2, 3`**.
3. **Evaluate the right hand sides**. Typically, this involves the evaluation of *expr* of the *assignment-stmt*. This is done for each member of the active set. Also at this stage is the evaluation of subexpressions on the left hand side (expression in the *variable* in the *assignment-stmt*). Again, these operations can be done in any order. In our example we have **`1./A(1),1./A(2),1./A(3)`**.
4. **Make assignment of left hand sides.** For each member of the active set make the assignment of the pre-calculated values (computed *expr* values) to the corresponding left hand sides of the expression (*variable* locations). Assignments may be in any order.

This process of execution is similar to that of array assignment, in that all the right hand sides are evaluated and then assignment is made to the left hand side. This is what we mean in saying that a **`FORALL`** is not a generalised parallel loop. The main point to note is that each stage of the execution must be completed before the next can begin (implicit synchronisation between stages), but within each stage the operations can be executed on the data elements in any order what so ever.

5.2.3 Visualising a FORALL

The standard way of visualising the execution of a **`FORALL`** statement is through a so called *precedence graph.* Such a graph shows all the computations performed in a **`FORALL`** and indicates where one computation must finish before another starts. Consider the following **`FORALL`** statement, with **`A`** with value **`[2 3 4 0 0 0]`**,

```
FORALL (i=1:3, A .NE. 0.0) B(i) = 1.0 / A(i)
```

Figure 16 shows the precedence graph for this example,

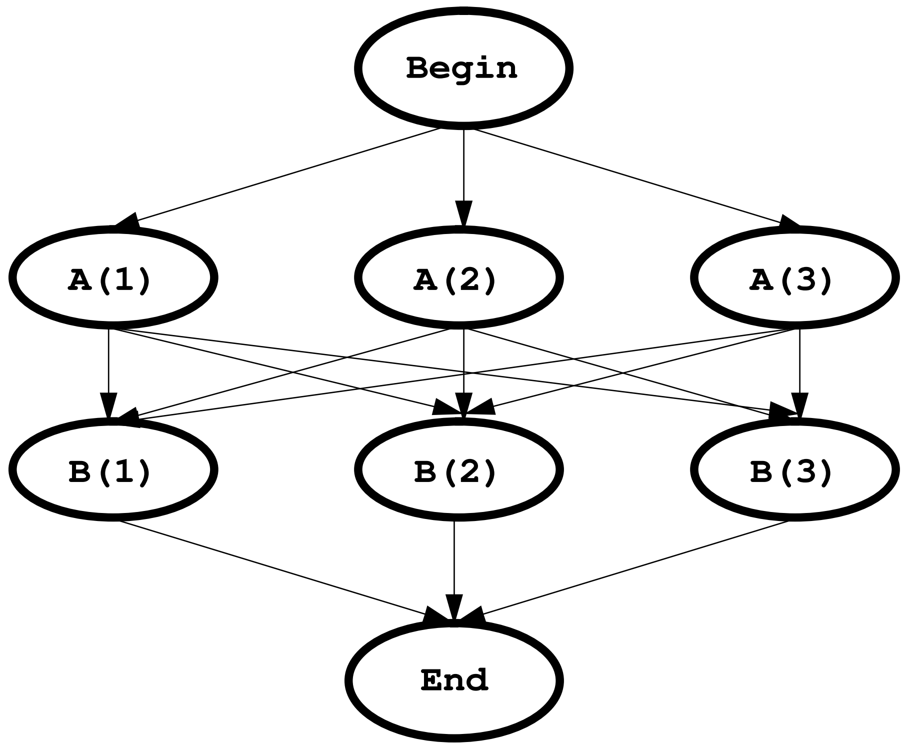


Figure 16: Precedence Graph for a FORALL Statement

In the precedence graph, the computations are shown as ellipses. The four stages of the **FORALL** are listed below:

1. The “Begin” denotes the computation of the **FORALL** active set. (This is in-fact two of the four stages: determining the valid index set, and evaluating the mask to produce the active index set)
2. Evaluation of the right hand side of the expression: the first line of ellipses takes the three elements of array **A**, and calculates their reciprocals (in any order).
3. Assignment to the left hand side: the reciprocals of elements of **A** (all now calculated) are assigned to the corresponding elements of **B** (in any order).
4. The “End” ellipse does not contain any computation, but indicates the end of the **FORALL** construct.

The arrows linking the ellipses do not indicate actual data dependencies, instead they indicate the order the computations may need to be executed. If a statement has to be done after another in a definite order, the arrow points from the earlier one to the later. The large number of arrows in Figure 16 is the “worst case” scenario, showing all possible orderings. In this case, these dependency arrows show that the execution of the **FORALL** synchronises at each step. All the evaluations of the right hand elements must be completed before assignment to the left hand side can begin. We shall see later how this synchronisation at each step can be overcome by use of the **INDEPENDENT** directive.

In an attempt to highlight the differences between the **FORALL** statement and a **DO** loop, we consider our previous example written using the **DO** construct.

```
DO i= 1, 3
   B(i) = 1.0 / A(i)
END DO
```

In a **DO** loop, the statements are executed in a strict sequential order. There is a point of synchronisation where the program waits for completion of one statement before starting the execution of the next statement. With the **FORALL** replaced by a **DO** loop in the previous program, we can visualise the execution of the program as shown in Figure 17.

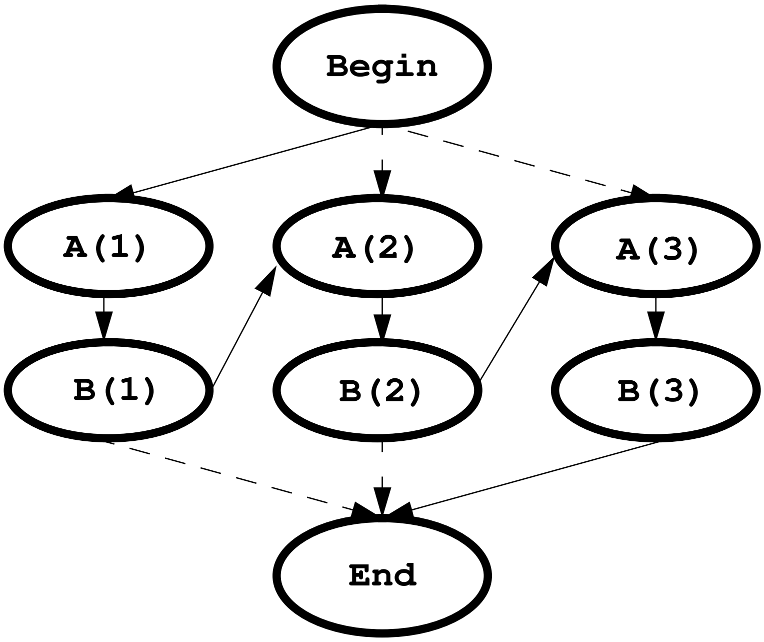


Figure 17: Example Of DO loop execution

The arrows again indicate the order of execution, the **DO** loop making one assignment after another. The full lines indicate the actual flow of control through the loop, whereas the dashed lines again just indicate the order of execution (so for example, the **DO** loop cannot "End" until assignments to **B(1)**, **B(2)** and **B(3)** have been made, but in practice this occurs after **B(3)** is assigned.

To further see the difference between the execution of a **DO** loop and a **FORALL** statement, consider the following program fragment with the initial values of the array **A** with **[3 2 0 1]**,

```
FORALL (I=2:4) A(I) = A(I-1)
```

Again we can visualise the **FORALL** statement by means of a precedence graph, shown in Figure 18,

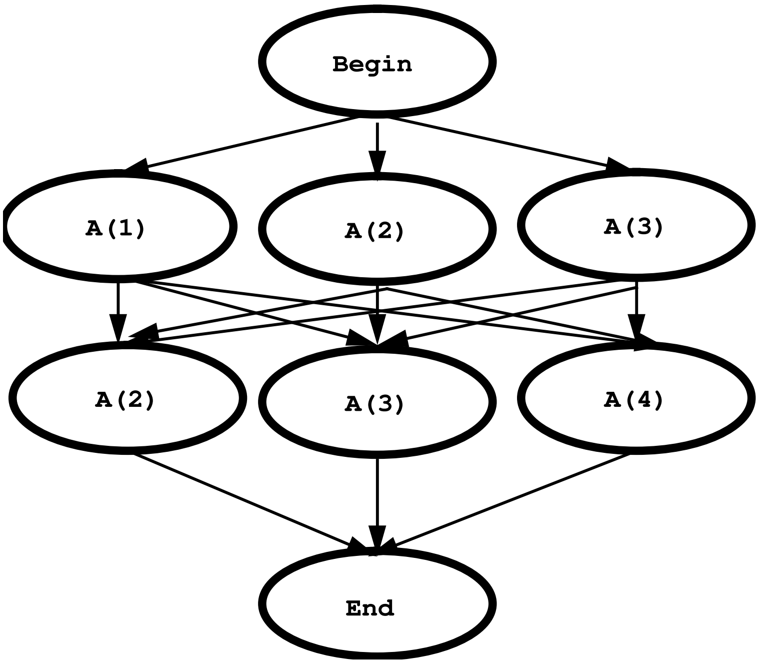


Figure 18: FORALL execution for the program fragment

The right hand side values in the **FORALL** construct are calculated simultaneously. Then there is a synchronisation (shown by the arrows between the first and second set of ellipses). With the final step the assignments to the left hand side. The value of array **A** after the **FORALL** is `[3 3 2 0]`.

If we execute the same program fragment with a **DO** loop then the end result will not be the same, with **A** becoming `[3 3 3 3]` if run in serial or an undefined result depending on the ordering of assignments if run in parallel. Clearly, in this case, the use of **FORALL** over a **DO** loop make a significant difference, especially if executed in parallel.

In Section 5.3 we will see how to give the compiler extra information about the ordering of instructions in a loop, that guarantees that no iteration of a loop will interfere with another.

5.2.4 FORALL Examples

To complete this section, we list a few examples which demonstrate the way **FORALL** can be used for assignments in a much neater way than possible with array statements only, though which preserve the inherit parallelism of an array expression not explicit in a **DO** loop.

1. HPF:

```
FORALL (I = 1:N, J = 1:M) A(I,J) = 1.0 / REAL( I + J )
```

   Fortran 90:

```
A = 1.0 / REAL( SPREAD((/( i,i=1,N)/), DIM=2, NCOPIES=M) + &
          +    SPREAD((/( j,j=1,N)/), DIM=1, NCOPIES=N) )
```

2. HPF:

```
FORALL(I=1:N, X(I).NE.0.0) Y(I) = 1.0 / X(I)
```

Fortran 90:

```
WHERE( X(1:N) .NE. 0.0) Y(1:N) = 1.0 / X(1:N)
```

3. HPF:

```
FORALL(I=1:N) A(I,I) = Y(I)
```

Fortran 90:

```
A = RESHAPE( MERGE((/(Y(i),(0,j=1,N),i=1,N-1),Y(N)/), &
       PACK(A,.TRUE.), (/((i==j,j=1,N),i=1,N)/)), (/N,N/) )
```

4. HPF:

```
FORALL(I=2:N-1) X(I) = X(I-1) + X(I) + X(I+1)
```

Fortran 90:

```
X(2:N-1) = X(1:N-2) + X(2:N-1) + X(3:N)

DO I = 2, N-1 !!! not the same !!!
       X(I) = X(I-1) + X(I) + X(I+1)
END DO
```

5.3 The INDEPENDENT Directive

The **INDEPENDENT** directive is used to provide the compiler with additional information about the execution of a **FORALL** construct or a **DO** loop. This is intended to allow the compiler to generate a parallel loop in situations where the parallelism is not obvious, but requires knowledge of the values of the data. It is a promise by the user to the compiler that the result of the loop statements will be the same whether executed in serial or in parallel. If the user breaks this promise and the iterations of the loop interfere with each other, then the code is not HPF conforming. It should be stressed that the **INDEPENDENT** directive should not alter the meaning of the code, it simply gives the compiler new information. This new information removes some restrictions from the compiler and allows it to generate more efficient code.

To see this more clearly, consider the following examples, based on an array assignments.

```
DO i= 2, 4
   A(i) = B(i-1)
END DO

FORALL (i = 1:2, A .NE. 0.0) B(i) = 1.0 / A(i)
```

In both examples the assignments can be made in parallel as there are no dependencies between the variables in different “iterations” of the loop. Clearly, these examples are very simplistic and one would hope a compiler would spot the lack of dependencies. However, by use of the **INDEPENDENT** directive the programmer can tell the compiler that the assignment **B(i) = 1.0 / A(i)** can be carried out in parallel, and that for any **i**, **A(i)** only depends on **B(i-1)** (which does not change in the loop) and each iteration assigns to a different element of **A**. In both examples, the answer will be same for parallel execution as obtained by sequential execution. The use of the **INDEPENDENT** directive in these examples is as follows, with the precedence graphs for each case shown, with the syntax of the directive as follows,

independent-directive is **INDEPENDENT** *[, new-clause]*

new-clause is **NEW** *(variable-list)*

,

```
DO i= 2, 4
  A(i) = B(i-1)
END DO
```

```
!HPF$ INDEPENDENT
      DO i= 2, 4
        A(i) = B(i-1)
      END DO
```

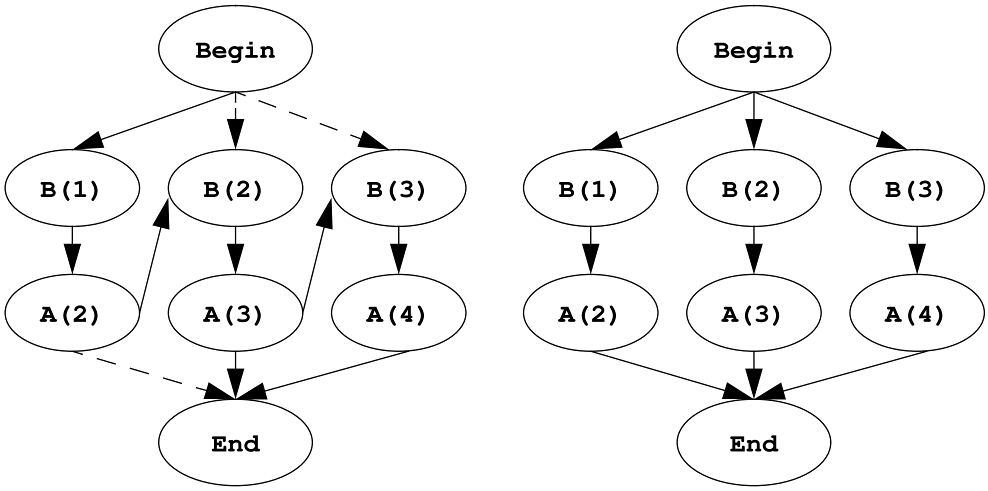


Figure 19: DO INDEPENDENT Precedence Graph

The effect of the **INDEPENDENT** directive on a **DO** loop is clear, each assignment in the loop can be done in parallel, as can be seen in Figure 19 from the reduced number of dependencies (arrows) when **INDEPENDENT** is used. A similar reduction is also seen in the **INDEPENDENT FORALL** construct in Figure 20. Although this **FORALL** statement could be parallelised without the use of the **INDEPENDENT** assertion, it is not possible to proceed onto the left hand side assignments until all the right hand side evaluations have been completed. This implicit synchronisation between steps in the execution of the **FORALL** can reduce the performance of the code.

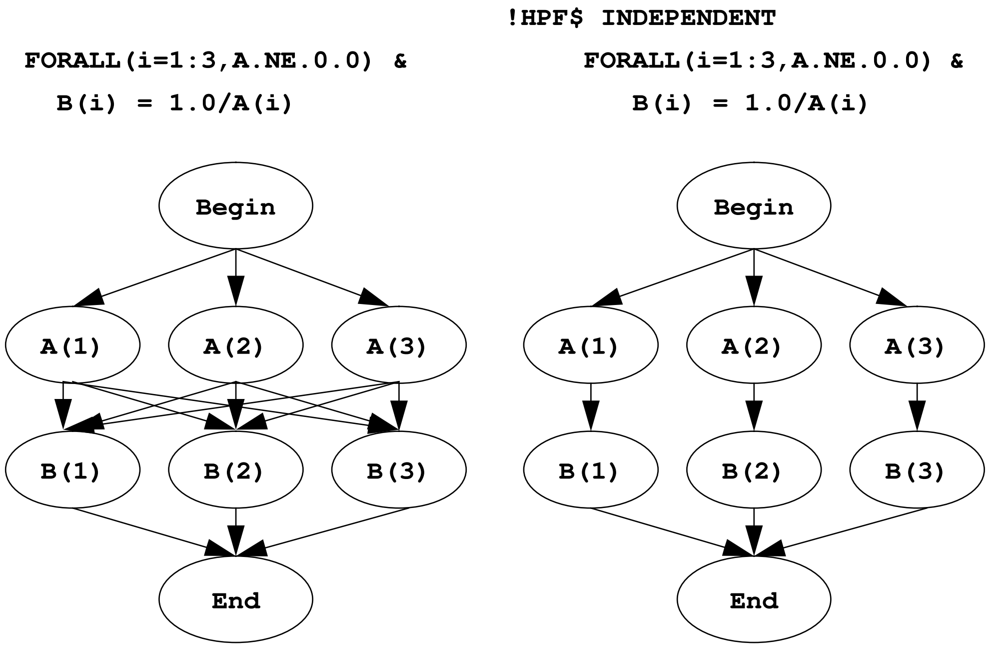


Figure 20: DO FORALL Precedence Graph

Suppose we are performing a set of assignments, across a set of processors. The amount of work needed for each assignment may not be the same, perhaps some communication is required. In Figure 21 we show a **`FORALL`** with two sets of assignments, in a graph where the height of the ellipse shows the length of time taken for that operation. So for example, the evaluation of **`RHS1(3)`** and **`RHS2(1)`** take longer. Also, in the graph without the **`INDPENDENT`** directive, we indicate synchronisation by solid bars. This synchronisation degrades performance in two ways. Firstly, the actual act of synchronisation has a time penalty. Secondly, with the **`INDEPENDENT`** directive, the algorithm need not progress in lock-step, so the execution stages of the **`FORALL`** can be overlapped.

```
! *NOT INDEPENDENT*
FORALL(i=1:3)
  LHS1(i) = RHS1(i)
  LHS2(i) = RHS2(i)
END FORALL
```

```
!HPF$ INDEPENDENT
   FORALL(i=1:3)
    LHS1(i) = RHS1(i)
    LHS2(i) = RHS2(i)
  END FORALL
```

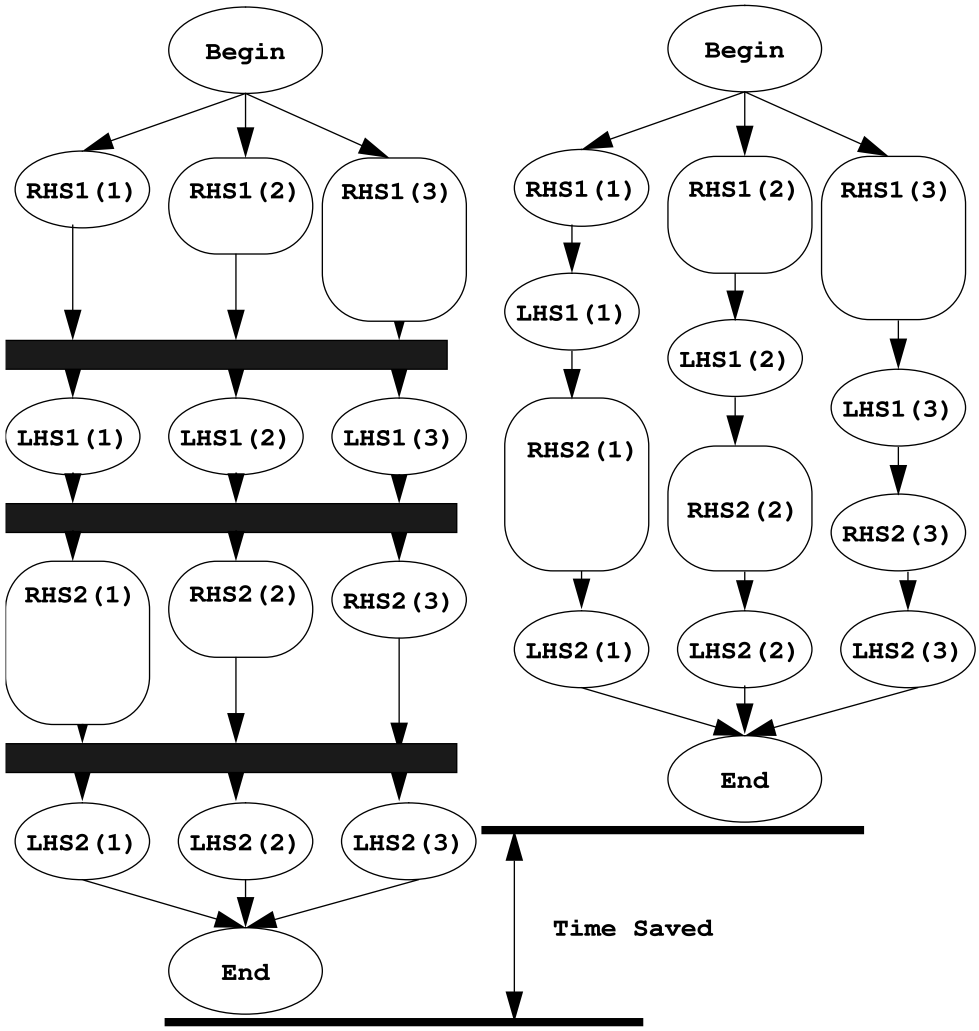


Figure 21: Avoiding Synchronisation in a FORALL by use of INDEPENDENT

The other main use of the **INDEPENDENT** directive is in situations where the compiler is not able to detect that the execution can be carried out in parallel. For example in the code below, when a vector subscript is used in an assignment, the **INDEPENDENT** directive asserts that there are no repeated values in the vector index, **INDX**.

```
!HPF$ INDEPENDENT
       DO i = 1, N
          A(INDX(i)) = B(i)
       END DO
```

A **DO** loop with an **INDEPENDENT** assertion applied to it is called a **DO INDEPENDENT** loop. Similarly, a **FORALL** construct with an **INDEPENDENT** assertion applied to it is called a **FORALL INDEPENDENT**.

Hopefully the above examples give a taste of the use of the **INDEPENDENT** directive in expressing the parallelism of a loop to the compiler. However, there are two important restrictions on use of this directive. The first involves the reading and writing of data objects and the second with control dependence in a loop. We will deal with each restriction in turn.

5.3.1 READ/WRITE Restrictions

In defining the restrictions on the reading and writing of data when used with the **INDEPENDENT** directive, it is useful to speak in terms of atomic data objects. An atomic data object is a data object which cannot be further broken down. For example, an array of integers is not an atomic data object, but one element of the array is an atomic data object.

The restrictions on the use of **INDEPENDENT** state that,

if a data object is written in one iteration, it cannot be written or read in another.

An example, which breaks this condition is given below,

```
DO i= 2, N          ! NOT INDEPENDENT
   A(i) = A(i-1)
END DO
```

This code fragment executes the operation of writing an atomic data object in one iteration and reading it in the next iteration. Clearly values of **A(i)** are dependent on **A(i-1)**. As a result, this code may produce different results when executed sequentially and in parallel, because values may be used which have not been updated by the assignment. In this case, therefore, a declaration of the **DO** loop as **INDEPENDENT** will be non-conforming in HPF.

The second restriction comes into play in the example of the calculation of a scalar product of two vectors,

```
S = 0.                     ! NOT INDEPENDENT
DO i= 1, N
   S = S + A(i) * B(i)
END DO
```

Although, there are no dependencies in the multiplication, the sum accumulated is read and written in each iteration. (A possible way round this is to do the multiplication into some scratch array, **D(i) = A(i)*B(i)**, and use the **SUM** reduction function to calculate the total, **S=SUM(D)**, or use HPF intrinsic function **DOT_PRODUCT**). In general it is not possible to use **DO INDEPENDENT** for accumulating operations, though HPF provides efficient parallel implementations of functions that do such operations.

These examples include very obvious, simplistic reads and writes. There are many ways in which the writing of an atomic data object occurs, such as,

- assignment statements,

```
A(j) = C        ! writes to A(j)
```

- using an object as a **DO** loop index, as a **FORALL** index, as input item in a **READ** statement, as an internal file in a **WRITE** statement. For example,

```
DO i= 1, N      ! Example of a write to i
FORALL (i = 1:N) ! Example of a write to i
READ *,i        ! i is written to in the READ
WRITE(i,*) N    ! i is internal file
```

Similarly an atomic data object can be read during,

- use of an atomic data object on the right hand side of an expression,

```
A(j) = C ! read of C
```

- Any I/O statement involving an atomic data object (includes I/O to a file)
- a **REALIGN** or **REDISTRIBUTE** directive (reads and writes every element of an array).

5.3.2 Control Dependence

The other restriction on an **INDEPENDENT** loop is that there must be no control dependence: if the statement starts to execute, it executes to completion. In other words the execution of the statements in a **DO INDEPENDENT** or **FORALL** construct cannot terminate due to a condition relating to one iteration.

For example, the code below searches for an array element,

```
SEARCH: DO i= 1, N
          IF( A(i) == 10 ) THEN
             EXIT SEARCH
          END IF
        END DO
        ...
```

This loop has control dependence because there are no further iterations after the required element is found. This control dependence is found in the execution of the **EXIT** statement. However, control statements, like **EXIT**, **STOP** or **PAUSE** can be included in a code if they are never executed i.e. if not values of A equal 10 (this code segment could also be independent if all values of A equal 10. In this case, the order of execution does not matter and there is no interference between iterations, because all iterations do exactly the same thing).

5.4 The NEW Clause

The **NEW** clause in an **INDEPENDENT** directive modifies the meaning of the directive by restricting the scope of the variables used. In other words, it is a means by which “temporary” variables can be made “private” to a loop. These temporary variables are allowed to by-pass some of the restrictions placed on the reading and writing of atomic data objects, as outlined above. An obvious example is in assignment using nested loops,

```
!HPF$ INDEPENDENT, NEW(i)
      DO j = 1, N
        DO i = 1, N
          A(i,j) = X(i,j) + &
            X(i+1,j) + X(i-1,j) + X(i,j+1) + X(i,j-1)
        END DO
      END DO
```

Here, variable **i** is written in each iteration in **j**. By including variable **i** in the **NEW** clause, the compiler is told that **i** should be treated as a new data object at each iteration, or that **i** loses its value after the loop. Such a problem does not occur in a **FORALL** statement as the index in this statement becomes undefined after the statement. Other examples include codes which use temporary variables as part of the calculation, as shown below,

```
!HPF$ INDEPENDENT, NEW(T)
      DO i = 1, N
        T = A(i) * B(i)
        A(i) = SQRT(T)
      END DO
```

5.5 The PURE Attribute

The **PURE** attribute is not in itself a parallel construct, but it is used to increase the generality of the **FORALL** statement. The main requirement on a procedure called from within a **FORALL** statement is that this procedure must not alter the execution of the **FORALL** loop. Use of the **PURE** attribute guarantees that the procedure will not have any side effects, and will not alter the input data in any way. Implicit in the use of the PURE attribute is that all input variables are specified **INTENT(IN)**. This means that **PURE** procedures are safe to use within **FORALL** statements.

All intrinsic functions are **PURE**. For example, **SIN()**, **COS()**, **EXP()** etc. can all be used in a **FORALL** statement. This cannot be said for all user defined functions. However, by use of the **PURE** attribute to qualify a procedure (used similarly to the **RECURSIVE** attribute in Fortran 90), the programmer can provide user-defined functions for use in a **FORALL** statement. For example, consider the following functions, all of which are **PURE**,

```
PURE REAL FUNCTION MY_EXP(X)
  REAL, INTENT(IN) :: X
  MY_EXP = 1 + X + X*X / 2.0 + X**3 / 6.0
END FUNCTION MY_EXP

PURE REAL FUNCTION MY_SINH(X)
```

```
  REAL, INTENT (IN) :: X
  MY_SINH = (MY_EXP(X) - MY_EXP(-X)) / 2.0
END FUNCTION MY_SINH

PURE REAL FUNCTION MY_COSH(X)
  REAL, INTENT(IN) :: X
  MY_COSH = (EXP(X) + EXP(-X)) / 2.0
END FUNCTION
```

Clearly, **MY_EXP** returns a value and does not alter the input data. The latter functions only call **PURE** functions, so are themselves **PURE**.

Another possible use for the **PURE** attribute is to provide functions for more complicated operations which can be done in parallel with a **FORALL** statement. For example, consider the Mandelbrot set exercise at the end of this section. In this example, the series of operations would not be allowed in a **FORALL** statement (assignment statements only are allowed). We would like to be able to use main routine such as the following,

```
FORALL(i = 1:N, j= 1:N)
   COLOUR(i,j) = MANDELBROT(Z_INITIAL(i,j))
END FORALL
```

for some initial condition, to determine **COLOUR** from the convergence of an iterative process. This can be done with the following function, which can be declared **PURE** because the dummy argument, **Z**, has been declared with **INTENT(IN)**, so cannot be modified (written) in this function, and no other side effects result.

```
          PURE FUNCTION MANDELBROT(Z)
             INTEGER              :: MANDELBROT
             COMPLEX, INTENT(IN)  :: Z
             COMPLEX              :: TMP
             INTEGER              :: i, COLOUR

             TMP = Z
ITERATE:     DO i= 0, 256
                IF (ABS(TMP) < 2.0) EXIT ITERATE
                TMP = TMP*TMP - Z
             END DO
             MANDELBROT = i

          END FUNCTION
```

This computation could not be included in the **FORALL** statement because of it contains a conditional and also it carries out an iterative procedure. However, by including the calculation in **PURE** function the **FORALL** can be used. The only other requirement is an **INTERFACE** block to inform the compiler of this function, of which a suitable version is shown below,

```
INTERFACE
   PURE FUNCTION MANDELBROT(Z)
      INTEGER :: MANDELBROT
      COMPLEX, INTENT(IN) :: Z
   END FUNCTION
END INTERFACE
```

It is hoped that these simple examples demonstrate the use of the **PURE** attribute in defining user functions suitable for use in a **FORALL** statement. However, we have not mentioned the many requirements placed on a function before it can be declared **PURE**. The programmer should consult the HPF standard for a full explanation of these restrictions.

The last point to note is that the **PURE** attribute is not in subset HPF.

5.6 Summary

In this chapter the **FORALL** construct and the **INDEPENDENT** directive were introduced. The **FORALL** construct is used to be specify operations that can be executed in parallel. The **INDEPENDENT** directive is used to tell the compiler that the **DO** loop or **FORALL** statement following the directive specifies operations that are independent and therefore can be performed in parallel. We also considered some examples to illustrate the use of these constructs.

5.7 Exercise 4: The Mandelbrot Set

The Mandelbrot Set is the set of numbers resulting from repeated iterations of the following complex function:

$$Z = Z^2 + C$$

which, separating real and imaginary form, looks like,

$$z_i = 2.0 * z_r \, z_i + c_i$$

$$z_r = {z_r}^2 - {z_i}^2 + c_r$$

for complex numbers, Z and C. This function is defined for complex values of $C=(c_r,c_i)$ in the range $([0.0,1.0], [0.0,1.0])$, with the initial conditions, $z=c$. In the case study, we will mainly be concerned with the Mandelbrot Set defined in the first quadrant of the complex plane, i.e. we will consider c in the range ([0.0, 1.0], [0.0,1.0]).

What is normally plotted is the number of iterations taken for z to reach some threshold value. We will take this threshold as

$$|z|^2 > 4.0$$

and set an upper iteration limit of 255. This number is converted into a greyscale or colour, and plotted at the point whose coordinates are (c_r, c_i).

5.7.1 FORTRAN 77 Code

Below is a fragment of a serial code which computes the Mandelbrot set. This should outline the algorithm to be used in the full HPF version.

```
c Declare N*N arrays for real and imaginary parts of
c arrays C and Z
c i.e. declare real arrays CR, CI, ZR, and ZIs
c     ...
c Initialise arrays CR, CI
c Initialise arrays ZR=CR, ZI=CI
c     ...
c Initialise ZIS and ZRS to hold the squares of ZR and ZI
c     ...
      DO i = 0, 255
        DO j = 1, N
          DO k = 1, N
            IF (ZRS(j,k) + ZIS(j,k) .LE. 4.0 ) THEN
              ZRS(j,k) = ZR(j,k) * ZR(j,k)
              ZIS(j,k) = ZI(j,k) * ZI(j,k)
              ZI(j,k) = 2.0 * ZR(j,k) * ZI(j,k) + CI(j,k)
              ZR(j,k) = ZRS(j,k) - ZIS(j,k) + CR(j,k)
              COLOUR(j,k) = i
            END IF
          END DO
        END DO
      END DO
c ...
```

5.7.2 HPF Version: Serial

Write an HPF program, based on the above code fragment, which will compute the Mandelbrot set, using the HPF/Fortran 90 array features to carry out the various stages (a template is available).

First off write a serial HPF version. A template is included with various pointers on how to proceed (and also the print statements required for a graphical output). The following steps are needed:

- **Initialisation**

  Initialise the *rows* of **CR** (real component of *C*) to be in the range [0.0,1.0] (i.e. every element in the first row has value 0.0, every element in the last row has value 1.0, and the intermediate rows have values varying between 0.0 and 1.0). Initialise the *columns* of **CI** (imaginary component of *C*) similarly. Use **FORALL** for this initialisation.

  Set the initial conditions **ZR** and **ZI** to be **CR** and **CI** respectively and define variables **ZRS** and **ZIS** to be the square of **ZR** and **ZI** respectively.

- **Iteration**

  Rewrite the above FORTRAN 77 code using Fortran 90 array syntax, using the **WHERE** construct.

  This program takes each point at a time and iterates the complex function (up to a maximum number of iterations, **RESOLUTION = 255**). The iterations stop when the absolute value of *Z* reaches or exceeds 2. The colour values are a measure of how many iterations it took to "escape" to 2 at each point.

- **Output**

  To write out the **COLOUR** array as a bitmap, use similar instructions as were used in the Game of Life. Include the following lines of code at the end of your program (this should already be in the template),

```
OPEN(UNIT=10,FILE='mandel.pgm')
WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 255
WRITE(10,*) colour
CLOSE(10)
```

Compile and run the code on a single workstation, with the size of the arrays set with **N = 128**. View the resulting bitmap with

```
xv -gamma 7 mandel.pgm
```

and check that the code works correctly (i.e. you recognise the bitmap to be the well known Mandelbrot set).

5.7.3 HPF Version: Parallel

Include the necessary HPF data mapping directives to distribute and align the arrays. Try both **BLOCK** and **CYCLIC** distributions. With this done, all the work is automatically distributed and the code "should" run in parallel.

Compile and run the code for 4 nodes. Check that the answer is still the same.

Use the profiling option to time the execution of the code and the amount of message passing involved.

Use this to compare how the performance is affected by use of different data distributions. Remember to comment out the input/output statements in the code while profiling.

6 Data Mapping

6.1 Objective

In this chapter we shall consider the HPF directives that control the mapping of distributed data. Roughly speaking, this is a two stage process based on the **DISTRIBUTE** and **ALIGN** directives. Data mapping also involves the **PROCESSORS** and **TEMPLATE** directives which are mentioned in this chapter. Throughout the following sections we attempt to illustrate the data mapping process, with pointers on how to produce efficient parallel code with the help of some examples and exercises.

We begin by outlining the data mapping model used by HPF, then introduce the data mapping directives and their syntax.

6.2 HPF Data Mapping Model

HPF directives allow the user to advise the compiler on allocation of data objects to processor memories. The mapping is performed in two stages. In the first stage, the group of data objects (such as arrays or array subsets) are aligned relative to one another, using the **ALIGN** or **REALIGN** directives. This group of aligned objects is then mapped onto a set of abstract processors, using the **DISTRIBUTE** or **REDISTRIBUTE** directives. The abstract processors are then mapped onto the real processors, with this mapping being done by the compiler in a system dependent manner.The data mapping model is shown in Figure 22.

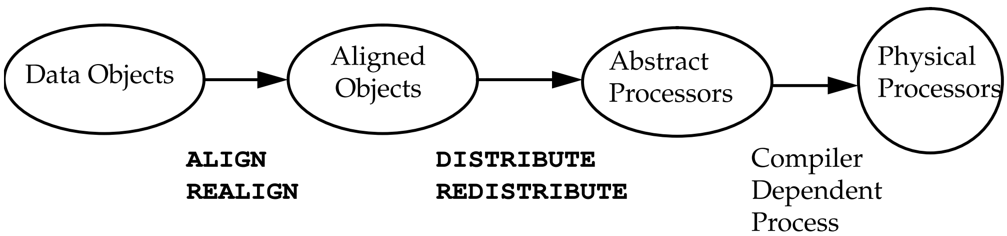


Figure 22: Data mapping model

With the **DISTRIBUTE** and **ALIGN** directives, the compiler generates the mappings at the start of the program. When the alignment or distribution changes with **REALIGN** or **REDISTRIBUTE** directives, a new set of aligned objects is generated which is then mapped onto the abstract processors arrangement, with the mapping onto the actual processors implemented by the compiler. In other words **DISTRIBUTE** and **ALIGN** are static and can only appear as program declarations, whereas **REDISTRIBUTE** and **REALIGN** are dynamic and can appear anywhere in the executable part of a program.

With this model of data mapping in mind, alignment can be considered to be an attribute of the data set. Initially an object is only aligned with itself. When the user wishes to express some interaction between objects, the original object gets aligned with the other objects or other objects get aligned to it. In this way we let the compiler know that we have a collection of objects which interact and therefore should be placed together on the abstract processors (so as to attempt to minimise communications). The object which is aligned with itself is called a target and it can be redistributed but not realigned. An object which is aligned with other object can be realigned during the execution of the program but not redistributed.

In discussing the distribution of data objects, we speak loosely of the array being distributed. Technically, distribution directives apply on the index space belonging to the array object. Distribution directives partition the index space onto the set of abstract processors. The combined operations of alignment and distribution of an array thus defines the relation of the array itself to the abstract processor, this operation being called the *'mapping'* of the array.

In the next section we attempt to motivate the need for distribution and alignment by means of a concrete example. A more detailed discussion of the directives follows this.

6.3 Data Mapping Overview

At present, compilers for parallel machines are not sophisticated enough to automatically distribute data arrays over a set of processors and obtain acceptable performance. In HPF there are directives to control the data mapping, which allow a user to specify the placement of data on the processors, thus giving the compiler more information to use to produce efficient code. Since the compiler generates communications during execution of the program, the only way to control the communication overhead is to control the data layout. If the user can keep pieces of closely interacting data elements (for example array sections) in close proximity on processor memory it is perhaps hoped possible to reduce the communications required.

To see this more clearly consider the following piece of FORTRAN 77 code, which does some simple image processing, and think about how this could be parallelised. This is typical of the kind of problems suited to a data parallel programming style: some kind of update algorithm on a regular grid using an array with a halo for the update.

```
      REAL centre(N,N), image(N+2,N+2)
      ...
      DO i = 1, N
        DO j = 1, N
          centre(i,j) = -
     &      -image(i,j)   - image(i,j+1)     - image(i,j+2)
     &     -image(i+1,j) + image(i+1,j+1)*8.- image(i+1,j+2)
     &     -image(i+2,j) - image(i+2,j+1)   - image(i+2,j+2)
        END DO
      END DO
```

Clearly, we could just run this code on a single processor of a multi-processor, but to make best use of the resources available we would like to run this code on a number of processors. In the HPF programming model, the way this is done is to spread the array elements across a number of processors, and for each processor to do some part of the work.

The first stage in the process is to tell the compiler about the abstract processor set available. This is done with the **PROCESSORS** directive. The next step of actually sharing the data across these abstract processors is through the **DISTRIBUTE** directive. In this particular example, we have two arrays which are offset with respect to each other, in that the calculation of **centre(i,j)** is related to **image(i+1,j+1)** and the surrounding points. We would like to be able to tell the compiler about this relationship in the hope that a these pieces of data could be kept on the same processor, reducing the communications required. HPF does this with the **ALIGN** directive.

This concludes the simple example of why and how we would want to use the **PROCESSORS, DISTRIBUTE** and **ALIGN** directives to map data arrays. In the following sections we go into more detail about these directives and mention the other HPF directives involved in data mapping.

6.4 The PROCESSORS Directive

In HPF, the arrangement of abstract processors may be specified by the **PROCESSORS** directive. The form of the directive is to give the arrangement a name and to specifying the extent of processors in each dimension. For example,

```
!HPF$ PROCESSORS example(6,2)
```

assumes there are 12 physical processors, and constructs a 6*2 array of abstract processors, which it names **EXAMPLE**. The actual syntax for this directive is as follows:

processors-directive	is	**PROCESSORS** *processors-decl-list*
processors-decl	is	*processors-name [(explicit-shape-spec-list)]*
processors-name	is	*object-name*

The rules for a **PROCESSORS** directive are:

- The *explicit-shape-spec-list* must contain the extents of all dimensions of the abstract processor array.
- The number of elements in the *explicit-shape-spec-list* must be the same as the rank of the array to distribute. So, for example, distributing a 2-D array needs a processor directive with a 2-D processor arrangement.
- The *processors-name* can not be same as a variable name, named constant or procedure name.
- Corresponding elements of a processor arrangement with the same shape refer to the same abstract processor. For example, with the following directives,

```
!HPF$ PROCESSORS P1(2)
!HPF$ PROCESSORS P2(2)
```

P1(1) and **P2(1)** refer to the same abstract processor.

- If two objects are mapped to the same abstract processor then the objects are mapped to the to same physical processor.
- If no shape is specified then the processor arrangement is scalar. For example,

```
!HPF$ PROCESSORS scalar
```

might be useful if the architecture has a master/slave processors or as a way of keeping scalar data together which is not intended to be aligned with distributed data.

It is possible to enquire from the system about the actual number of processors at run time and their shape, using the intrinsic functions **NUMBER_OF_PROCESSORS()** and **PROCESSOR_SHAPE()**. These functions can also be used inside the abstract processor arrangement specification, for example,

```
!HPF$ PROCESSORS grid(NUMBER_OF_PROCESSORS()/3,3)
```

The HPF compiler is required to accept any declaration where the product of extents in the processor arrangement is the same as the number returned by the **NUMBER_OF_PROCESSORS()** function.

The above use of the **PROCESSORS** directive defines the arrangement of a set of abstract processors. The actual mapping of this set of abstract processors to a set of physical processors is not defined in HPF.

6.5 The DISTRIBUTE Directive

In Section 6 we briefly outlined the use of the **DISTRIBUTE** directive to map array indices to abstract processors as part of the HPF data mapping process. In this section we will look at this directive in more detail and indicate the different possible mappings available. First we begin with a few concrete examples, and then finish this section with the full syntax of the directive.

For example, suppose we have four abstract processors and a two dimensional array **A(100,100)**. Then the following code fragment will distribute the array such that each processor will contain 2500 elements. The distribution of array elements is shown in Figure 23, along with an indication of which array elements may rest on which abstract processor (in array notation).

```
!HPF$ PROCESSORS procs(2,2)
      INTEGER, DIMENSION(100,100) :: A
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO procs :: A
```

Block distributions such as this are good for problems which have a regular domain decomposition, such as fluid dynamics and Quantum Chromodynamics.

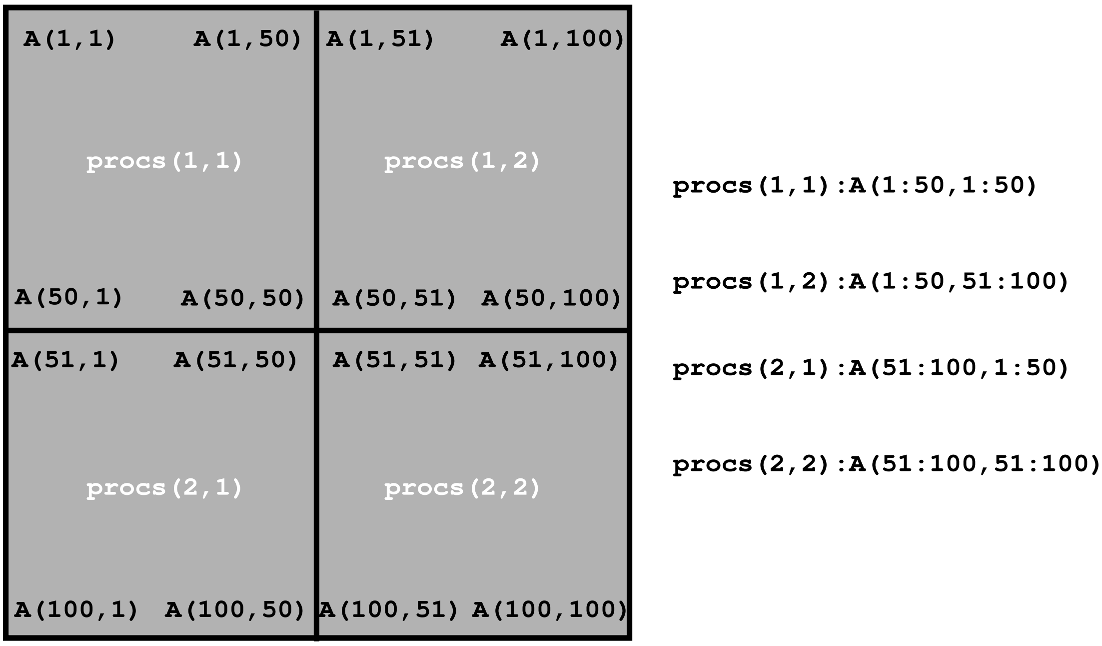


Figure 23: Two Dimensional Block Distribution

It is also possible to apply a block distribution along one axis only. In this case we specify,

```
!HPF$ PROCESSORS procs(4)
!HPF$ DISTRIBUTE (BLOCK,*) ONTO procs :: A
```

then, the distribution of the array will be along the rows as shown in Figure 24. When we specify a dimension in an array by '`*`' then that dimension is not considered while distributing the array. In the case shown below an entire row of array **A** is distributed as a single object (which is completely local), and then these objects are distributed among the abstract processors. We say that this axis has been “collapsed”, and we denote the distribution labelled by a **`*`** as a degenerate distribution.

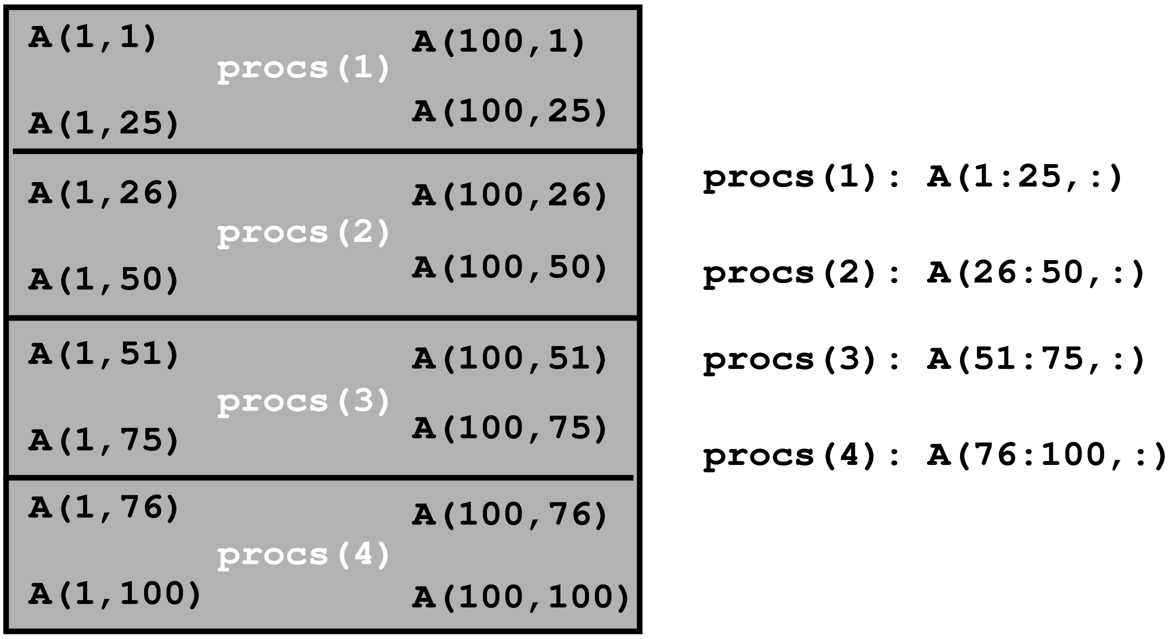


*Figure 24: (BLOCK, *) distribution*

The size of a block can also be specified explicitly in the block distribution. For example,

```
!HPF$ PROCESSORS, DIMENSION(p) :: procs
!HPF$ DISTRIBUTE (BLOCK(10),*) ONTO procs :: A
```

will have one block of 10 elements of **`A`** on each processor (of course, this assumes there are sufficient processors, otherwise the statement does not conform to HPF. If there are **`p`** processors and **`d`** is the number of data elements in the dimension of the array, then for the block distribution statement to be HPF conforming, the block size **`m`** must be such that, **`m*p`** **`d`**The default block size, **`BLOCK = BLOCK(m)`**, is given by block size **`m =`** $\lceil$**`d/p`**$\rceil$, which describes the integer upper bound on the devision. If the block size is specified to be too small, the resulting distribution is not defined by HPF. So, for example, if we have 100 data elements (**`d=100`**) and 16 processors (**`p = 16`**), then **`BLOCK(7)`** and **`BLOCK(8)`** are defined, but **`BLOCK(6)`** is HPF non-conforming.

There is another type of distribution known as the cyclic distribution. In this case the array elements are cyclically distributed to each processor. Figure 25 gives an example of a cyclic distribution for a one dimensional array **`C`** corresponding to the following code.

```
!HPF$ PROCESSORS procs(4)
      INTEGER, DIMENSION(1000) :: C
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: C
```

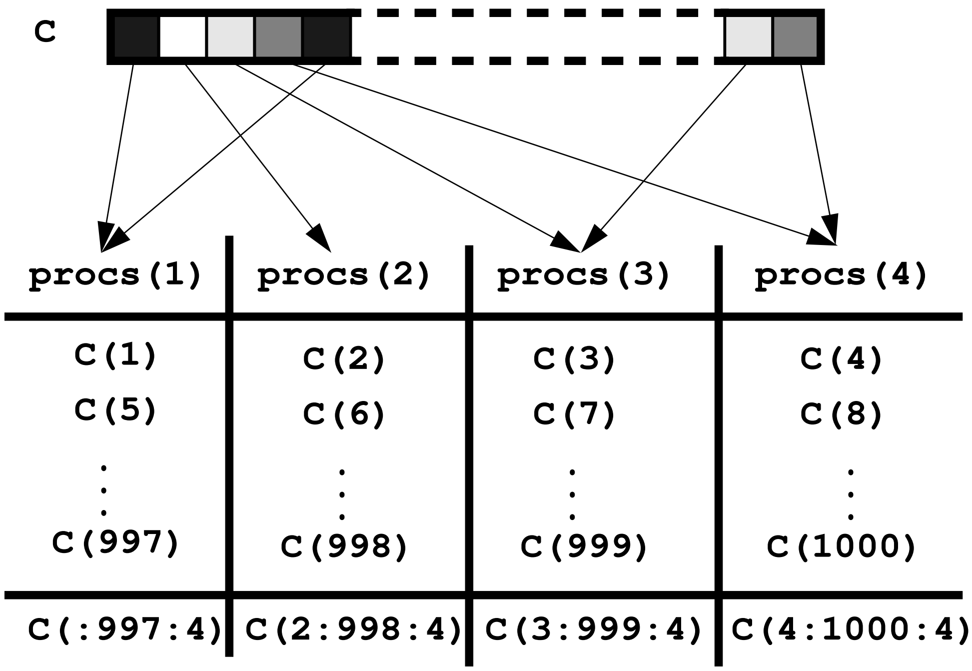


Figure 25: Cyclic distribution

The assignment of array elements to processors is indicated for the first eight elements. The cyclic distribution can be thought of as a round-robin allocation of processors to array elements: **`C(1)`** to processor 1, **`C(2)`** to processor 2, etc. Cyclic distributions are good for problems where the locality of data interactions is not so important, but where the work required for each section of the array varies significantly. Examples of this include ray tracing, edge detection and gaussian elimination.

Similarly, the cyclic distribution can be given an explicit block size. For example, **`C(CYCLIC(5))`**, would block together 5 elements of array **`c`**, with these blocks distributed cyclically. There is no restriction on the cyclic block size. The default definition is **`CYCLIC = CYCLIC(1)`**. The **`CYCLIC`** and **`BLOCK`** distributions are equivalent if $p \geq d$, that is, there are sufficient processors for each data object to sit on a unique processor. More generally, the distributions **`BLOCK(m)`** and **`CYCLIC(m)`** are equivalent if $m \times p \geq d$, for a data object with dimension **`d`** distributed over **`p`** processors.

Many complex mappings can be generated from these directives. Distributions should be chosen which keep processors busy and minimise communication.

6.5.1 DISTRIBUTE Directive Syntax

In this section we list the formal syntax of a **`DISTRIBUTE`** directive as given in the HPF Specification v1.0.

distribute-directive	is	**`DISTRIBUTE`** *distributee dist-directive-stuff*
dist-directive-stuff	is	*dist-format-clause [dist-onto-clause]*

dist-attribute-stuff	is	*dist-directive-stuff*
	or	*dist-onto-clause*
distributee	is	*object-name*
	or	*template-name*
dist-format-clause	is	*(dist-format-list)*
	or	* *(dist-format-list)*
	or	*
dist-format	is	**BLOCK** *[(int-expr)]*
	or	**CYCLIC** *[(int-expr)]*
	or	*
dist-onto-clause	is	**ONTO** *dist-target*
dist-target	is	*processors-name*
	or	* *processors-name*
	or	*

with the constraints that:

- the object to be distributed (the *distributee*) must be a simple name (e.g. array name).
- the *distributee* may not appear as an *alignee* in an **ALIGN** statement (i.e. it may be aligned to, but may not itself be aligned).
- a *dist-format-list* must have as many elements as the rank of the *distributee* (i.e. every dimension of an array must be distributed in some way).
- if both a *dist-format-list* and *processors-name* are present, then the number of elements in the *dist-format-list* (that are not a *) must be equal to the rank of the processor arrangement. (e.g. distributing a 2-D array **(BLOCK,BLOCK)** requires a 2-D processor arrangement; distributing a 2-D array **(BLOCK,*)** requires a 1-D processor arrangement)
- if the *dist-format-clause* or the *dist-target* begin with a *, then every *distributee* must be a dummy argument

6.6 The ALIGN Directive

Sometimes it is convenient to express a desired distribution for an array by describing its relationship with respect to some other array. The **ALIGN** directive can be used to specify such a relationship. An example of this was seen earlier, where it was useful to keep related array sections on the same processor. The aim of this alignment was to produce a more efficient code, with reduced communications.

As an illustration consider the problem of the Game of Life, encountered in an earlier exercise. Here we have two arrays, **board** and **neighbours**, say, which are used together under some algorithm. The distribution of these two arrays could be specified with two **DISTRIBUTE** directives as follows

```
!HPF$ DISTRIBUTE board(BLOCK,BLOCK) ONTO procs
!HPF$ DISTRIBUTE neighbours(BLOCK,BLOCK) ONTO procs
```

In this case the alignment between these two arrays is implicit, and you would expect corresponding elements of these arrays to exist on the same processor. This works for a simple example, but in general it is clearer to explicitly align the arrays using the **ALIGN** directive:

```
!HPF$ DISTRIBUTE board(BLOCK,BLOCK) ONTO procs
!HPF$ ALIGN neighbours WITH board
```

which guarantees the alignment between the elements of the two arrays. The added advantage of this, is that this relationship defined by the alignment remains when the distribution is changed (a great saving if many arrays are involved).

Returning to our image processing example introduced at the start of this section, used to motivate the use of distribution and alignment. This is the FORTRAN 77 code for the update algorithm used.

```
      REAL centre(N,N), image(N+2,N+2)
      ...
      DO i = 1, N
        DO j = 1, N
          centre(i,j) =
   &     -image(i  ,j) - image(i  ,j+1)     - image(i  ,j+2)
   &     -image(i+1,j) - image(i+1,j+1)*8.0 - image(i+1,j+2)
   &     -image(i+2,j) - image(i+2,j+1)     - image(i+2,j+2)
        END DO
      END DO
```

Note that (i,j) th component of **centre** is obtained from 3x3 square of index pairs around $(i+1,j+1)$ in **image**. In porting this to HPF, we could distribute **image** over, say, 4 processors, using the following directives

```
!HPF$ PROCESSORS square(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square::image
```

where it is sensible to use a **BLOCK** distribution because the interactions between arrays is nearest-neighbour only (which is almost optimally bad for **CYCLIC** distributions). The relationship between these arrays can be expressed with the following directive:

```
!HPF$ ALIGN WITH image(i+1,j+1) :: centre(i,j)
```

This alignment is shown in Figure 26, where the inner array represents centre and the outer, image.

.

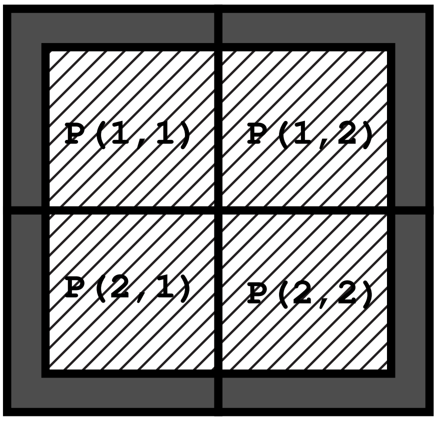


Figure 26: Alignment of arrays

When these directive are included with Fortran 90 style array syntax, the resulting HPF code looks like:

```
      REAL, DIMENSION(N,N) :: centre
      REAL, DIMENSION(N+2,N+2) :: image
!HPF$ PROCESSOR square(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: image
!HPF$ ALIGN WITH image(i+1,j+1) :: centre(i,j)
      ...
      centre(:,:) =
     & -image( :N,  :N)-image( :N,  2:N+1)   -image( :N,  3:N+2)
     & -image(2:N+1,:N)-image(2:N+1,2:N+1)*8 -image(2:N+1,3:N+2)
     & -image(3:N+2,:N)-image(3:N+2,2:N+1)   -image(3:N+2,3:N+2)
```

6.6.1 Collapsing an Array using ALIGN

Like the **DISTRIBUTE** directive, **ALIGN** allows for collapsing of a dimension of the array. For example,

```
      REAL B(4,3) M(4)
!HPF$ ALIGN B(i,*) WITH M(i)
```

will result into the alignment as shown in the Figure 27, where the double arrows indicate that the objects are stored on the same processor i.e. they are aligned in memory.

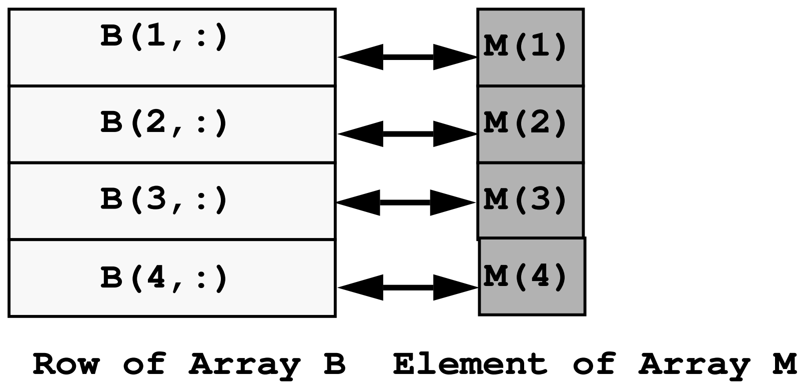


Figure 27: Collapsing Alignment

This would mean for example that the vector sum corresponding to the following

```
M = B(:,1) + B(:,2) + B(:,3)
```

would generate no communication. Once again, if the distribution of **`M`** changed then this would not affect the alignment. Equivalent forms of the align statement above are:

```
!HPF$ ALIGN B(:,*) WITH M(:)
!HPF$ ALIGN B(i,k) WITH M(i)
```

The use of dummy variables or "**`:`**" or "**`*`**" are equivalent in these statements, but the latter two are often used to give stronger visual clues to the meaning of the alignment.

6.6.2 Alignment for Replication

The **`ALIGN`** directive can be used to replicate heavily used arrays, such as lookup tables. By placing a copy of this replicated array on each processor it may be possible to avoid much communication with other processors accessing the lookup table. In some sense this is the converse operation to collapsing alignment outlined above. Consider the following code fragment, in which we suppose that the array M is used in some way by operations on every element of A:

```
      INTEGER M(4)
      INTEGER A(4,5)
!HPF$ ALIGN M(*) WITH A(i,*)
```

This directs the compiler to align **M** with every row of **A**. Essentially this means that the compiler must store a copy of **M** in every processor that **A** is distributed on, trading off the extra storage requirements for savings in communications. This is illustrated graphically in Figure 28, where again, the double arrows indicate alignment of objects in memory.

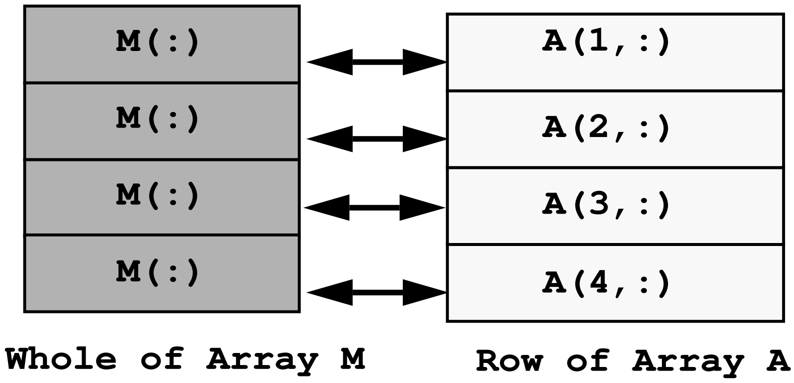
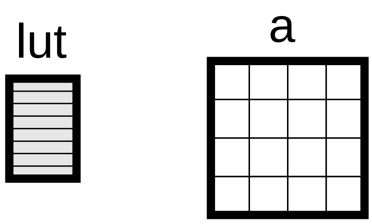


Figure 28: Replication for Alignment

When **M** is updated the compiler ensures consistency by updating all copies of the array. If the compiler is clever enough, it would not store a copy of the array for each element it is aligned to, rather it would store one copy of the replicated array on each processor.

For example, consider the following example of a lookup table:




```
REAL a(4,4)
REAL lut(8)
...
FORALL (i=1:4, j=1:4) a(i,j) = a(i,j) + lut(i+j)
```

The following line would align the *whole* of the lookup table *with every* element of **a**:

```
!HPF$ ALIGN WITH a(*,*) :: lut(*)
```

as shown in Figure 29.

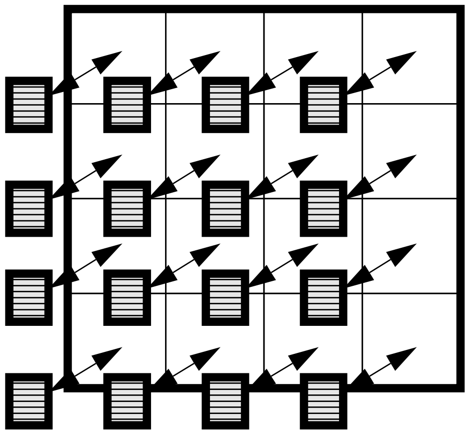


Figure 29: Replicated lookup table

If the processor arrangement was specified as a 2*2 square of processors, you might be able to presumed an implementation with distribution as shown in Figure 30,

```
!HPF$ PROCESSORS square(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: a
```

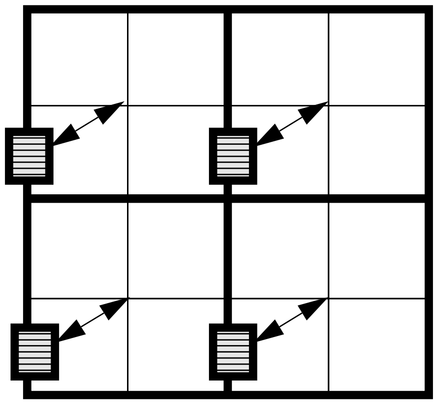


Figure 30: Presumed implementation on four processors

6.6.3 Example: Simple Matrix Multiplication

A widely used operation in scientific computing is matrix multiplication, multiplying two matrices **`A`** and **`B`** to form a matrix **`C`**. An essential component of the matrix multiplication is the ability to multiply the **`i`**th row of **`A`** by the **`j`**th column of **`B`** to form the element **`C(i,j)`**. To perform the multiplication effectively on a multiprocessor machine, it is essential that we have the ability to keep rows of **`A`** and columns of **`B`** on the same processor. One way of doing it is use distribution directives as given below.

```
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: C
!HPF$ DISTRIBUTE (*,BLOCK) ONTO line :: A
!HPF$ DISTRIBUTE (BLOCK,*) ONTO line :: B
```

There are two problems with this. The fact that rows of **A** and columns of **B** are tied together is not reflected in these statements. If one of the distributions was changed then the resulting code would be communication intensive, as this change would not be reflected in a change of the other distribution. Also, unless the **ONTO** clause is specified then there is no guarantee that rows of **A** and columns of **B** a will be aligned as we wish. Also if there are many arrays interacting together, then specifying the distributions as given above results in clumsy looking code.

The **ALIGN** directive allows the programmer to specify the distribution of one array by describing its alignment with another array. The following code specifies the relationships we desire for **A**, **B** and **C**.

```
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: C
!HPF$ ALIGN A(i,*) WITH C(i,j)
!HPF$ ALIGN B(*,j) WITH C(i,j)
```

Note that if the distribution of **C** changes, then the distribution of **A** and **B** will also change to keep the alignment. Also, this guarantees that the rows of **A** and the columns of **B** are stored together.

Using these directives we can now code our multiplication program. In this example we use a simple algorithm, focusing on the distribution and alignment issues.

```
      PROGRAM ABmult
      IMPLICIT NONE
      INTEGER, PARAMETER :: N = 100
      INTEGER, DIMENSION (N,N) :: A, B, C
      INTEGER :: i, j

!HPF$ PROCSSORS square(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: C

!HPF$ ALIGN A(i,*) WITH C(i,j)
!       replicate copies of row A(i,*)
!       onto processors which compute C(i,j)

!HPF$ ALIGN B(*,j) WITH C(i,j)
!       replicate copies of column B(*,j))
!       onto processors which compute C(i,j)

      A = 1
      B = 2
      C = 0
```

```
      DO i = 1, N
        DO j = 1, N
!         All the work is local due to ALIGNs
          C(i,j) = DOT_PRODUCT(A(i,:), B(:,j))
        END DO
      END DO

      WRITE(*,*) C

      END
```

This program aligns the rows of **A** and columns of **B** as we require, but not explicitly. By broadcasting copies of the relevant rows/columns to where they are used, the rows of A and columns of B are aligned implicitly.

6.6.4 Align Syntax

Here we list the formal syntax of the **ALIGN** directive as listed in the HPF Specification v1.0.

align-directive	is	**ALIGN** *alignee align-directive-stuff*
align-directive-stuff	is	*(align-source-list) align-with-clause]*
align-attribute-stuff	is	*[(align-source-list)] align-with-clause*
alignee	is	*object-name*
align-source	is	:
	or	*
	or	*align-dummy*
align-dummy	is	*scalar-int-variable*
align-with-clause	is	**WITH** *align-spec*
align-spec	is	*align-target [(align-subscript-list)]*
	or	** align-target [(align-subscript-list)]*
align-target	is	*object-name*
	or	*template-name*
align-subscript	is	*int-expr*
	or	*subscript-triplet*
	or	*

The main constraints on this are as follows:

- the *alignee* may no be the *distributee* in a **DISTRIBUTE** directive (i.e. an array to be aligned with another, should not already be distributed)

- if the *align-spec* begins with a *. then the *alignee* should be dummy variables (only for use in procedures)

The following forms are equivalent:

```
!HPF$ ALIGN alignee (align-source-list) WITH align-spec
!HPF$ ALIGN (align-source-list) WITH align-spec :: alignee
```

6.7 The TEMPLATE Directive

Templates are useful for providing a large index space against which smaller objects may be aligned, without actually allocating memory for the real array. They are simply objects which span an index space (arrays of nothing as opposed to the arrays of integers etc.) and as such take up no storage space. The directive specifies a name and an extent for the template, the full syntax of which is

template-directive	is	**TEMPLATE** *template-decl-list*
template-decl	is	*template-name [(explicit-shape-spec-list)]*
template-name	is	*object-name*

For a more concrete example, consider the following template

```
!HPF$ TEMPLATE A(100)
```

declaring an index space of size 100, but not actually allocating the memory required for storage of an actual array.

Templates are particularly useful, for example, when we wish to align four smaller arrays of size **N** by **N** with the four corners of a larger array of size **N+1** by **N+1**, but without actually declaring a full-sized array,

```
!HPF$ TEMPLATE, DIMENSION(N+1,N+1) :: WORLD
      REAL, DIMENSION (N,N) :: NW, NE, SW, SE
!HPF$ ALIGN NW(i,j) WITH WORLD(i  , j  )
!HPF$ ALIGN NE(i,j) WITH WORLD(i  , j+1)
!HPF$ ALIGN SW(i,j) WITH WORLD(i+1, j  )
!HPF$ ALIGN SE(i,j) WITH WORLD(i+1, j+1)
```

Note that the declaration of the template **WORLD** does not cause any memory allocation as it is an index space. The relationships between the arrays **NW** and **SE** and the index space are illustrated below.

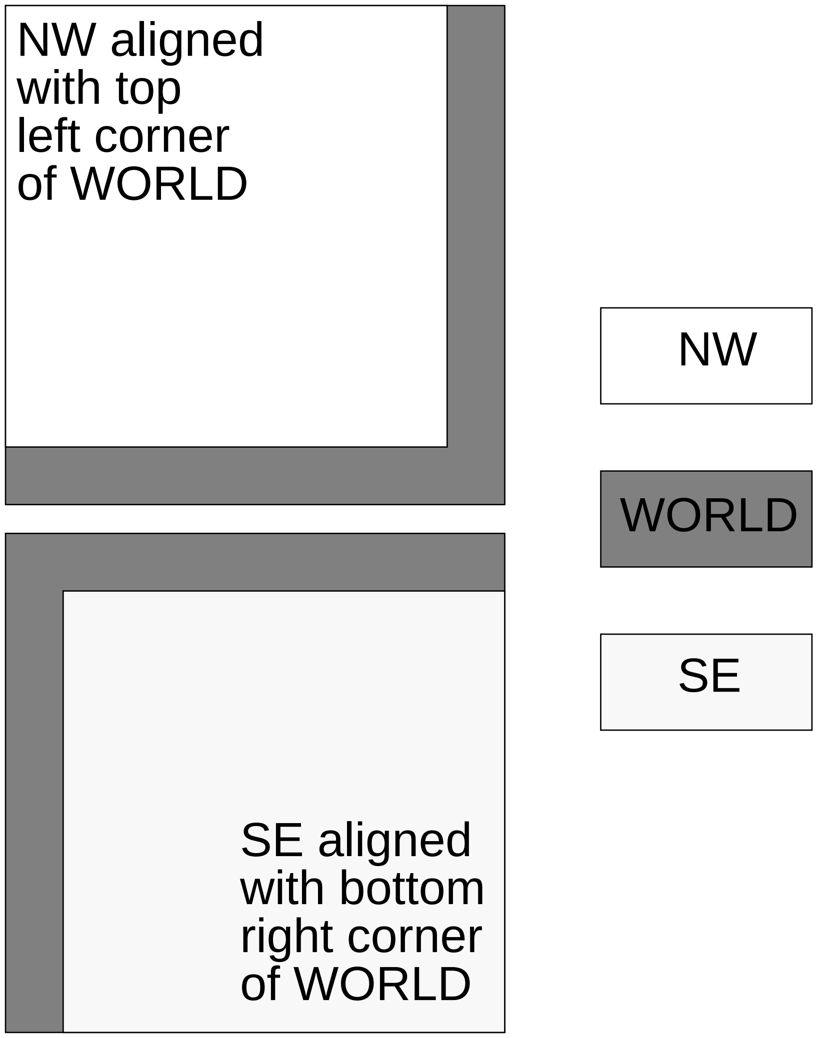


Figure 31: Alignment using the TEMPLATE directive

6.8 Dynamic Data Mapping

In the sections above we have introduced the issues of data mapping using the **DISTRIBUTE**, **ALIGN** and **PROCESSORS** directives. These are all declarations and must appear in specification part of the program module. However, there are a set of directives in HPF which allow the data mapping to change during the execution of the program.

These directives are **REALIGN** and **REDISTRIBUTE**, the dynamic forms of the **ALIGN** and **DISTRIBUTE** directives respectively. These directives operate in a similar fashion, although they can only exist in the execution part of the module. Both of these directives require that the argument must have been declared with the **DYNAMIC** attribute. The **REDISTRIBUTE** directive will cause any array to be redistributed at any time, and will also ensure that any aligned arrays will also be redistributed. Similarly the **REALIGN** directive will change the alignment of an array at any time. The main restriction to this is that the arrays to be redistributed or realigned must have been declared with the **DYNAMIC** directive.

In the following example, the alignment of **A1** would stay the same but its distribution would change with that of **D2**. **A2** would be realigned with **D1**.

```
!HPF$ DISTRIBUTE D1(CYCLIC)
!HPF$ DISTRIBUTE D2(BLOCK), DYNAMIC
!HPF$ DYNAMIC, ALIGN WITH D2 :: A1, A2
...
!HPF$ REDISTRIBUTE D2(CYCLIC(7))
...
!HPF$ REALIGN A2 WITH D1
```

Further details of these statements are available in the standards documentation.

However, it should be noted that dynamic data mapping is not in subset HPF.

6.9 Summary

In this chapter we have introduced the data mapping capabilities of HPF. In particular we have concentrated on the following directives,

- **PROCESSORS**
- **DISTRIBUTE**
- **ALIGN**
- **TEMPLATE**
- **DYNAMIC / REDISTRIBUTE / REALIGN**

However, in concentrating on these main topics in data mapping, we have had to omit several topics which were outside the scope of this course. In particular we have made no mention to storage association in HPF (the use of **COMMON** and **EQUIVALENCE**) and the **SEQUENCE** / **NO SEQUENCE** directives used to advise the compiler on whether to treat the data objects as sequential or not. This topic is covered briefly in Section 9, the Advanced Features section. For further detail, we leave the reader to consult the HPF standard or any of the numerous text books on the subject.

The other main omission to this chapter has been the treatment of data mapping issues in relation to procedure arguments. This was given a separate chapter and is dealt with next.

In summary, we have attempted to demonstrate the flexibility and power of the HPF data mapping model.

6.10 Exercise 5: Birthday

The purpose of this exercise is to choose appropriate **DISTRIBUTE** and **ALIGN** directives for a particular problem.

Our example is that of finding the day of a year any given individual's birthday falls within a particular (non-leap) year. A simple way to do this is to produce a look-up table with the number of days in each month (or more appropriately, a running total of these values), which is consulted for every birthday. The day on which the birthday falls is the sum of the total number of days so far plus the number of days in that particular month.

The following code defines data structures and data initialisation along with a print statement useful for viewing the results (a template is provided with code already written).

```
      PROGRAM birthdays
      IMPLICIT NONE
      INTEGER, PARAMETER :: n=1000
      INTEGER, DIMENSION(n,2) :: dob
      INTEGER, DIMENSION(n)   :: days
      REAL, DIMENSION (n) :: rand
      INTEGER i
      INTEGER, DIMENSION(12) :: days_in_month,days_so_far
      DATA days_in_month/31,28,31,30,31,30,31,
     &              31,30,31,30,31/
!     ...
!     Fill in this part
!     ...
      PRINT 1,dob(:15,1),dob(:15,2),days(:15)
 1    FORMAT(' DOB: month : ',15i4/
     &       ' DOB:   day : ',15i4/
     &       '       days : ',15i4/)
      END
```

The array, **days_in_month**, is initialised to contain the number of days in each month. You will also find it useful to produce another array, **days_so_far**, which contains a cumulative total of the number of days in each month.

The array **dob** is used to hold each individual's date of birth with **dob(:,1)** holding the months and **dob(:,2)** the days within the month. Initalise this to contain a random set of birthdays, using the Fortran 90 intrinsic subroutine, **RANDOM_NUMBER**. to create a set of random numbers in the range [0,1] in array **rand**.

Your task is to complete the program and calculate the number of days from the beginning of the year for each birthday. The result should be assigned to the array **days**.

Firstly decide which arrays should be distributed and how. Use a mixture of **DISTRIBUTE** and **ALIGN** directives to do this.

Pay particular attention to the alignment of the array **days_in_month/days_so_far**.

Compare the performance of the code when run on a single and multiple workstations, and also when the distribution statements are included.

7 Procedure Arguments and Data Mapping

7.1 Introduction

Procedures are essential components of a modular programming style. Invoking a procedure establishes a correspondence between the actual arguments of the calling routine and the dummy arguments defined in the called routine. In earlier chapters we discussed the directives included in HPF which define how data objects (the actual arguments) may be mapped onto multiple processors. In this chapter we discuss the directives provided by HPF for data mapping of dummy arguments in procedures.

The distribution of dummy arguments in subroutines is different from the distribution we have met so far because of the large number of possibilities. For example,

- sometimes the data needs the data with a specific distribution (and will remap the data if necessary)
- sometimes the subroutine does not need to know the distribution of the incoming data (and no remapping is required)
- sometimes the programmer knows the distribution of the incoming data, and sometimes not.

The main idea behind data mapping of procedure arguments in HPF is that a subprogram should not permanently change the mapping of a data object: the distribution must be the same before and after a subprogram call (this restriction allowing the compiler to generate more efficient code). However, it is possible for data to be remapped on entry to a subroutine. For example, consider a subroutine which might be more efficient for a particular mapping of the input arrays and hence remapping of the arrays on entry to the subroutine may be desirable and is also necessary for the subroutine to work with arrays of arbitrary mapping. However, remapping of an array is likely to be communication intensive and hence should be minimised. For this reason, HPF does not *require* that remapping takes place.

7.2 Directives for Data Mapping

There are three types of mapping that can be defined for dummy arguments in a procedure; prescriptive, descriptive and transcriptive.

With a prescriptive mapping the programmer advises the compiler about the mapping of procedure arguments, perhaps for efficiency of an algorithm on data objects with a particular mapping.

With a descriptive mapping the programmer assures the compiler about the data mapping of procedure arguments, providing a means to obtain maximal efficiency of code by asserting to the compiler that the data will be in a particular form.

With the transcriptive mapping, procedure arguments inherit the mappings from the data mappings in the caller routines, allowing a subprogram to deal with data objects however they may be distributed, but at the cost of reduced performance from increased generality. In all these cases, the compiler may or may not need to physically remap the data.

In all of these cases, if the mapping directive of the caller does not satisfy that of the callee, then an implicit remapping must occur. This, however, is not visible to the caller routine as everything must be distributed as before after a routine has been called. Also, it is possible to use Fortran 90 style interface blocks to provide remapping information at compile time, again to allow for the generation of more efficient code. If an interface block is not included, the code is still HPF conforming if the caller and callee mapping directives agree or if the callee allows remapping.

In an attempt to make this more clear we will consider concrete examples of each type of argument mapping. There are four possibilities to consider when a distributed object is passed to a callee routine,

1. the distribution of the data object in both caller and callee is identical,
2. the distribution of the data object in the caller and callee is different,
3. the distribution of the data object in the caller and callee is different and the interface is explicit,
4. the distribution of the data object in the caller and callee is the same and the interface is explicit.

In each case we consider arrays distributed over a set of four abstract processors in either **(BLOCK,BLOCK)** or **(CYCLIC,CYCLIC)** distributions. We include a fragment of code for the set of caller routines and also the callee procedure, indicating syntax of the argument distribution directives and the necessity of implicit remapping or not.

7.2.1 Prescriptive Argument Mapping

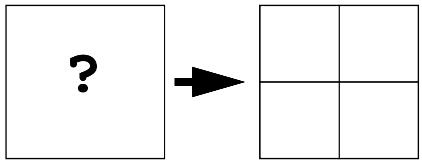
In a prescriptive directive, the mapping of data elements is prescribed in the code of the callee procedure and optionally through an interface block in the caller procedure. These mappings are directives to the compiler to re-map the data, if required, to produce the data mappings prescribed in the callee procedure. Hence the programmer has control over the mapping of the data elements in a callee procedure. The prescriptive mapping is useful, when a programmer knows what the data layout will be or alternatively needs a particular layout for efficiency reasons.

The form of a prescriptive mapping statements for dummy arguments is exactly the same as met earlier:

```
!HPF$ DISTRIBUTE C(BLOCK,BLOCK)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) :: C
```

An example of a subroutine that asserts a prescriptive mapping is shown in for subroutine **CATCHA.**

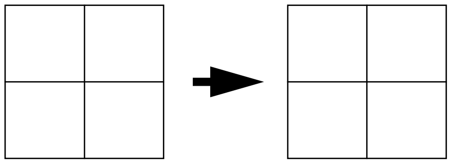
```
      SUBROUTINE CATCHA(C)
      REAL C(10,10)
!HPF$ DISTRIBUTE C(BLOCK,BLOCK)
      ...
      END
```



The four cases outlined in the previous section for prescriptive mapping are shown in subroutines **EANY**, **MEANY**, **MINY** and **MO**

1. In subroutine **MEANY**, the distribution of array **B** is the same as in routine **CATCHA**, so no remapping is required, though there may be some time penalty for testing whether remapping is required.

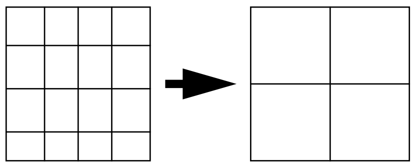
```
      SUBROUTINE MEANY
      REAL B(10,10)
!HPF$ DISTRIBUTE B(BLOCK,BLOCK)
      ...
      CALL CATCHA(B)
      ...
      END
```



No remapping required

2. In passing array **A** to **CATCHA** from **EANY**, the data must be implicitly remapped. The actual array has distribution **(CYCLIC,CYCLIC)** which must be remapped to **(BLOCK,BLOCK)**. On return from **CATCHA**, array **A** is remapped again to maintain the **(CYCLIC,CYCLIC)** distribution.

```
      SUBROUTINE EANY
      REAL A(10,10)
!HPF$ DISTRIBUTE A(CYCLIC,CYCLIC)
      ...
      CALL CATCHA(A)
      ...
      END
```

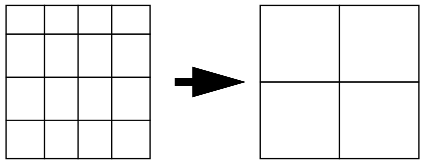


Implicit Remapping

3. Subroutine **MINY** contains arrays with the same distribution as in **EANY**, requiring implicit remapping, however, by inclusion of the **INTERFACE** block the compiler knows at compiler time that a remapping is required and it can attempt to produce an efficient remapping. Again, array **C** has its original

`(CYCLIC,CYCLIC)` distribution on return from **CATCHA**.

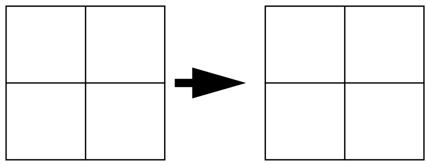
```
      SUBROUTINE MINY
      REAL C(10,10)
!HPF$ DISTRIBUTE C(CYCLIC,CYCLIC)
      INTERFACE
        SUBROUTINE CATCHA(P)
          REAL P(10,10)
!HPF$      DISTRIBUTE P(BLOCK,BLOCK)
        END SUBROUTINE
      END INTERFACE
      ...
      CALL CATCHA(C)
      ...
      END
```



Implicit Remapping (Known at compile time)

4. With the distribution of data in subroutine **MO** being the same as that in **CATCHA** and with the inclusion of an **INTERFACE** block nothing need be done, as the compiler knows at compile time that no remapping is required.

```
      SUBROUTINE MO
      REAL D(10,10)
!HPF$ DISTRIBUTE D(BLOCK,BLOCK)
      INTERFACE
        SUBROUTINE CATCHA(P)
          REAL P(10,10)
!HPF$      DISTRIBUTE P(BLOCK,BLOCK)
        END SUBROUTINE
      END INTERFACE
      ...
      CALL CATCHA(D)
      ...
      END
```



No remapping required (Known at compile time)

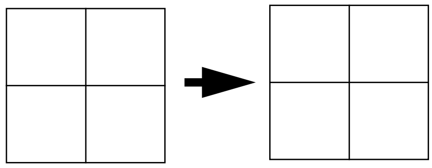
7.2.2 Descriptive Argument Mapping

With a descriptive data mapping the programmer *describes* the data mapping in the callee procedure and assures the compiler that the callee will only be passed data with that particular mapping. Therefore, no data remapping is required at run-time if the arguments are passed by reference. The requirement that no data remapping is needed is signified by an asterisk in the **DISTRIBUTE** directive for the dummy argument, i.e

```
!HPF$ DISTRIBUTE S *(BLOCK,BLOCK)
!HPF$ DISTRIBUTE *(BLOCK,BLOCK) :: S
```

An example of such a subroutine is shown below as subroutine **STONES**, where dummy argument **S** is required to have a **(BLOCK,BLOCK)** distribution..

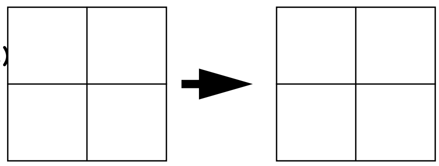
```
      SUBROUTINE STONES(S)
      REAL S(10,10)
!HPF$ DISTRIBUTE S *(BLOCK,BLOCK)
      ...
      END
```



Again there are four cases for descriptive data mapping. These are shown below in code segments in subroutines **EMERALD**, **RUBY**, **AMBER**, **SAPPHIRE**.

1. The code for the second case is contained in subroutine **EMERALD**. In this case there is a match between the distribution of the actual and dummy arguments, so nothing need be done in passing array **GREEN** to **STONES**.

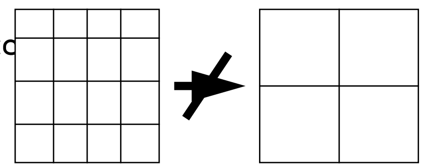
```
      SUBROUTINE EMERALD
      REAL GREEN(10,10)
!HPF$ DISTRIBUTE GREEN(BLOCK,BLOCK)
      ...
      CALL STONES(GREEN)
      ...
      END
```



No remapping required

2. Array **RED** in subroutine **RUBY** is distributed in a **(CYCLIC,CYCLIC)** form. This does not match the distribution specified in the callee subroutine for the mapping of dummy argument **D**. As a result, the call is HPF nonconforming and the program behaviour is undefined in HPF.

```
      SUBROUTINE RUBY
      REAL RED(10,10)
!HPF$ DISTRIBUTE RED(CYCLIC,CYCLIC
      ...
      CALL STONES(RED)
      ...
      END
```

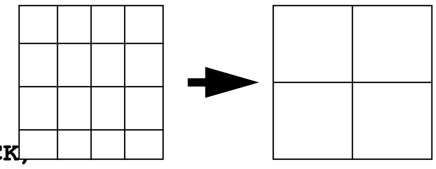


Nonconforming call

3. In the third case, in subroutine **AMBER**, we have the actual array distributed as **(CYCLIC,CYCLIC)**. When the array **YELLOW** is passed to subroutine **STONES** it must be implicitly remapped. Compared to subroutine **RUBY**, this case is HPF conforming because **AMBER** uses an **INTERFACE** block describing the descriptive mapping directives of **STONES**. In this case a descriptive mapping con-

tained in an **INTERFACE** block should be treated as a prescriptive mapping.

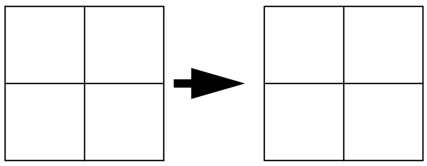
```
      SUBROUTINE AMBER
      REAL YELLOW(10,10)
!HPF$ DISTRIBUTE YELLOW(CYCLIC,CYCLIC)
      INTERFACE
        SUBROUTINE STONES(S)
          REAL S(10,10)
!HPF$     DISTRIBUTE S *(BLOCK,BLOCK)
        END SUBROUTINE
      END INTERFACE
      ...
      CALL STONES(YELLOW)
      ...
      END
```



Implicit Remapping (Known at compile time)

4. In the last case, described in subroutine **SAPPHIRE**, the distributions of the actual and dummy arguments are the same, so nothing need be done. Specifying the descriptive directive in an **INTERFACE** statement tells the compiler of the agreement of the mapping of the actual and dummy arguments at compile time.

```
      SUBROUTINE SAPPHIRE
      REAL BLUE(10,10)
!HPF$ DISTRIBUTE BLUE(BLOCK,BLOCK)
      INTERFACE
        SUBROUTINE STONES(S)
          REAL S(10,10)
!HPF$     DISTRIBUTE S *(BLOCK,BLOCK)
        END SUBROUTINE
      END INTERFACE
      ...
      CALL STONES(BLUE)
      ...
      END
```



No remapping required (Known at compile time)

7.2.3 Transcriptive Argument Declarations

In a transcriptive argument mapping, the data distribution properties for data elements are transcribed (in other words, followed as they are) from the caller procedure. The transcriptive mapping is useful when a callee procedure is intended to be used with data that may be mapped in any way. In other words, in a transcriptive directive the actual arguments are never remapped.

Note, transcriptive mapping is not in subset HPF.

This is done by use of the following directive

```
!HPF$ INHERIT D
```

which implies a **DISTRIBUTE** directive of the form

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

The **DISTRIBUTE** directive specifies that the data distribution is *as is*, from the caller procedure, whereas the **INHERIT** directive specifies that the template of the dummy is to be inherited from the corresponding element in the caller procedure. There is no need to have the **DISTRIBUTE** directive when the **INHERIT** directive is present.

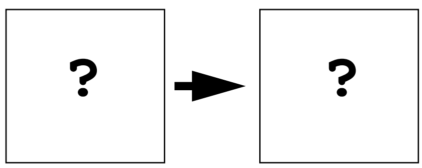
The resulting code, although it avoids having to remap arrays, may be slightly slower as it has to cater for all different data distributions in the callee procedure. However, it is more versatile in that it allows the programmer to determine the mapping of the dummy argument and then make a decision on the action to be taken.

For example, if the programmer wanted to write a general subroutine to do some operation, which could be performed with a number of different algorithms depending on the mapping of the input data. Using a transcriptive mapping, the programmer can use the HPF intrinsic to determine the distribution of the dummy argument and then call the appropriate subroutine which executes the algorithm for that particular distribution. This is demonstrated in the following code, which uses user-defined functions (**CYCLIC** and **BLOCK**) to determine the actual mapping of the input data. The subroutines **CYCLIC_GAUSSIAN** and **BLOCK_GAUSSIAN** could then use descriptive mappings:

```
SUBROUTINE GAUSSIAN(A)
  REAL, DIMENSION(:,:) :: A
  INHERIT A
  IF (CYCLIC(A)) THEN
    CALL CYCLIC_GAUSSIAN(A)
  ELSE IF (BLOCK(A)) THEN
    CALL BLOCK_GAUSSIAN(A)
  ELSE
    CALL DEGENERATE(A)
  END IF
END SUBROUTINE
```

An example of a transcriptive mapping is given in subroutine **DOBEEDO** below, showing that any distribution of the dummy argument can be accepted without remapping.

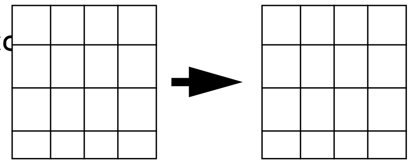
```
      SUBROUTINE DOBEEDO(D)
      REAL D(10,10)
!HPF$ DISTRIBUTE D * ONTO *
!HPF$ INHERIT D
      ...
      END
```



In the case of a transcriptive mapping, the four possibilities are shown below in subroutines **SCOOBY**, **SHAGGY**, **THELMA** and **DAPHNE**.

1. In subroutine **SCOOBY**, array **A** is distributed in **(CYCLIC,CYCLIC)**. No remapping is required, though subroutine **DOBEEDO** must be prepared to accept its dummy argument with distribution **(CYCLIC,CYCLIC)**.

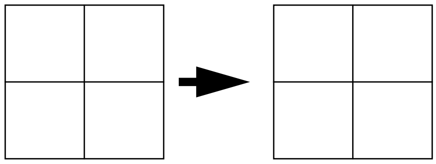
```
      SUBROUTINE SCOOBY
      REAL A(10,10)
!HPF$ DISTRIBUTE RED(CYCLIC,CYCLIC)
      ...
      CALL DOBEEDO(A)
      ...
      END
```



No remapping occurs

2. Subroutine **SHAGGY** passes array **B** with distribution to **DOBEEDO** with no remapping, with **DOBEEDO** prepared to accept the dummy argument in **(BLOCK,BLOCK)** distribution.

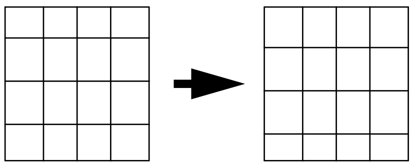
```
      SUBROUTINE SHAGGY
      REAL B(10,10)
!HPF$ DISTRIBUTE B(BLOCK,BLOCK)
      ...
      CALL DOBEEDO(B)
      ...
      END
```



No remapping occurs

3. Similar to **SCOOBY**, subroutine **THELMA** has **(CYCLIC,CYCLIC)** distribution and no remapping is needed. However, the inclusion of an **INTERFACE** block allows the compiler to generate more efficient code for the call.

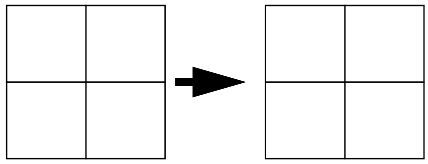
```
      SUBROUTINE THELMA
      REAL C(10,10)
!HPF$ DISTRIBUTE C(CYCLIC,CYCLIC)
      INTERFACE
        SUBROUTINE DOBEEDO(D)
          REAL D(10,10)
!HPF$     DISTRIBUTE D * ONTO *
!HPF$     INHERIT D
        END SUBROUTINE
      END INTERFACE
      ...
      CALL DOBEEDO(C)
      ...
      END
```



No remapping occurs
(Known at compile time)

4. The final case, subroutine **DAPHNE**, again requires no remapping and the Fortran 90 **INTERFACE** block again provides additional information for the compiler.

```
      SUBROUTINE DAPHNE
      REAL E(10,10)
!HPF$ DISTRIBUTE E(BLOCK,BLOCK)
      INTERFACE
        SUBROUTINE DOBEEDO(D)
          REAL D(10,10)
!HPF$     DISTRIBUTE D * ONTO *
!HPF$     INHERIT D
        END SUBROUTINE
      END INTERFACE
      ...
      CALL DOBEEDO(E)
      ...
      END
```



No remapping occurs (Known at compile time)

Again, we note that transcriptive mappings are not included in subset HPF.

7.2.4 Distribution and Processor Arrangement

As a last point, we note that in the above we have been concentrating on the different types of formats, prescriptive, descriptive and transcriptive, of the **DISTRIBUTION** directive. Prescriptive, descriptive and transcriptive definitions also exist for the processor arrangement (the **ONTO** clause in a **DISTRIBUTE** directive) allowing for a number of possible combinations.

So for example,

```
!HPF$ PROCESSORS two(2)
!HPF$ DISTRIBUTE * ONTO two
```

would be a prescriptive mapping on the processors arrangement (and transcriptive on the distribution), and similarly,

```
!HPF$ PROCESSORS four(4)
!HPF$ DISTRIBUTE * ONTO *four
```

would be a descriptive mapping onto the processors arrangement (signified by the *). This would mean that the subroutine expects the incoming data to be arranged on a set of 4 abstract processors (and the code is non-conforming, if this is not the case, unless an interface block explicitly states the usage).

7.2.5 Alignment

The final stage in the HPF data mapping model is the alignment of arrays. This too has prescriptive and descriptive forms, with the transcriptive using the **INHERIT** directive.

So for example,

```
      SUBROUTINE SIZE(LONG,SHORT)
      REAL, DIMENSION(1000) :: LONG, SHORT
!HPF$ INHERIT :: SHORT
!HPF$ ALIGN WITH SHORT :: LONG
```

would be a prescriptive alignment on array **LONG** with array **SHORT**, whose ultimate alignment and distribution was **INHERIT**ed. Similarly,

```
      SUBROUTINE WEIGHT(HEAVY,LIGHT)
      REAL, DIMENSION(1000) :: HEAVY, LIGHT
!HPF$ ALIGN WITH *HEAVY :: LIGHT
```

would assume array **LIGHT** was arriving already **ALIGN**ed with **HEAVY**.

7.3 Dummy Arguments and Templates

We have seen in the data mapping model of HPF, that the mapping of a data object may be defined either using the **DISTRIBUTE** directive, or by its relation to another data object as specified by the **ALIGN** directive. We now introduce the concept of a template of ultimate alignment. The template of a dummy argument is determined from the original arguments by the following rules.

1. If the dummy argument is explicitly **ALIGN**ed to another data object, then the template of the dummy argument is the *template* of the data object it is aligned to. For example:

   ```
         SUBROUTINE CALLEE(DUMMY)
   !HPF$    ALIGN DUMMY WITH TEMP1
   ```

   specifies that the array **DUMMY** is to be aligned with **TEMP1**. Hence the template of the **DUMMY** is the same as that of the array **TEMP1**.
2. If the dummy argument is not explicitly aligned to another data object (as above) and does not have the **INHERIT** attribute, then a new template is created, called the natural template, which has the same size and shape as the dummy argument. The dummy is then aligned with itself.
3. If the dummy argument is not explicitly aligned to another data object but has an **INHERIT** attribute, then the template for the dummy argument is inherited as follows:
   - If the data argument is a whole array , then the template of the dummy

is a copy of the template to which the original data argument is aligned.

- If the data argument is a regular array section, then the template of the dummy is a copy of the template to which the original data argument is aligned.
- If the actual argument is any other expression then a freshly created template is used, the shape and distribution of which may be chosen arbitrarily by the language processor .

The template found in this way is known as the inherited template.

For example,

```
      INTEGER, DIMENSION(100) :: CINDY, NAIOMI, LINDA
!HPF$ PROCESSORS CATWALK(8)
!HPF$ DISTRIBUTE (BLOCK) ONTO CATWALK ::CINDY,NAIOMI,LINDA
      CALL FASHION(CINDY(1:50:2), NAIOMI(1:50:2), &
         LINDA(1:50:2))
      ...

      SUBROUTINE FASHION(MODEL1, MODEL2, MODEL3)
      INTEGER MODEL1(:), MODEL2(:), MODEL3(:)
      INTEGER, DIMENSION(25) :: YSL
!HPF$ INHERIT MODEL3
!HPF$ ALIGN MODEL1 WITH YSL
      ...
      END SUBROUTINE CALLEE
```

In this example we have all three cases of templates for dummy arguments. In the **FASHION** subroutine, **MODEL1** is **ALIGN**ed to another data object (**YSL** of shape **[25]**). Dummy argument **MODEL2** is not explicitly aligned to another data object and it does not have an **INHERIT** attribute, hence the template for **MODEL2** is of shape **[25]**, the same size as **MODEL2** itself. However, **MODEL3** inherits its template from the actual argument in the caller routine. In this case, **MODEL3** has a full 100 element template and **MODEL3(I)** is aligned with **2*I-1** element of the template and still has a **BLOCK** distribution. This becomes important when the **FASHION** subroutine attempts to make a descriptive distribution on the dummy arguments. Including the line

```
!HPF$ DISTRIBUTE MODEL3 *(BLOCK)
```

would be correct, as the actual argument is already in **BLOCK** distribution with a template of shape [100] (same as for the actual argument). However, it would not be correct to use

```
!HPF$ DISTRIBUTE MODEL2 *(BLOCK) ! *** Non conforming ***
```

because the shape of the template for **MODEL2** is **[25]**. This means that the layout of the actual elements of actual argument **NAIOMI(1:50:2)** cannot be properly distributed in **BLOCK** over a template of **[25]**. A way around this, however, is to align the dummy argument with a template of the correct size by, for example, including the following lines in subroutine **FASHION**,

```
!HPF$ PROCESSORS CATWALK(8)
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK) ONTO CATWALK :: WESTWOOD
!HPF$ ALIGN MODEL2(I) WITH *WESTWOOD(2*I-1)
```

Advised not to use array sections as procedure arguments. Instead it is safest and most easily understood to pass whole arrays to procedures.

7.4 Summary

In this chapter, we have considered the data mapping of procedure arguments. There are three forms of data mapping for procedure arguments; prescriptive, descriptive and transcriptive. The salient features of each of these mappings are:

- In the prescriptive form, the data mapping in the procedure argument is specified to the compiler. Actual re-mapping of the data may or may not take place according to the specifications. The form of the directive is for example,

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

- In the descriptive form, the data mapping in a procedure argument is described and an assurance is given to the compiler that it will be as described at run time.

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

- In the transcriptive form, the data distribution properties of the procedure arguments are inherited from the calling procedure. In a transcriptive form, the same procedure can be used with different caller routines and with different data distributions of the procedure arguments.

```
!HPF$ DISTRIBUTE * :: A
```

The rules for determining the template for a dummy argument were discussed, followed by an example.

7.5 Exercise 6: Life in a subroutine

The aim of this exercise is to rewrite the distributed Game of Life from Exercise 3, with the iterations of the update of the system done in a subroutine.

Before doing this, include all the data mapping constructs you have met so far (i.e. set the **PROCESSORS** directive and use **ALIGN** to specify the relationship between the board and its neighbours).

The subroutine should take the board array and the number of iterations as inputs. Within the subroutine, the update should be performed and the result printed to file as before. All iterations of the update should be done in this subroutine.

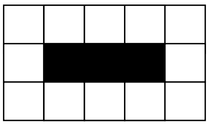
- Start with a descriptive mapping. Copy the distribute directives for the actual arrays in the main program into the subroutine (to distribute the dummy arguments in the same way).
- Generalise to a prescriptive mapping (distribute the actual arguments with a different mapping than the dummy arguments)

In each case, you should include an interface block for each subroutine, in the main program.

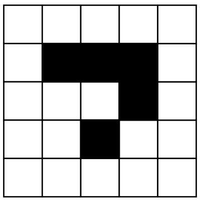
Compare the performance of these two cases, trying to see the overhead introduced by remapping in the prescriptive case.

7.5.1 Extra Exercise

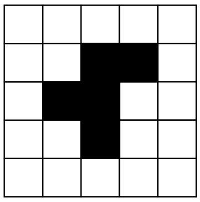
Try some of the following patterns as initial conditions for you board:



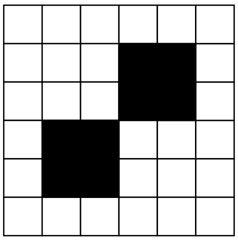
Blinker



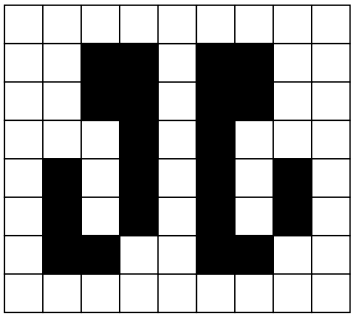
Glider



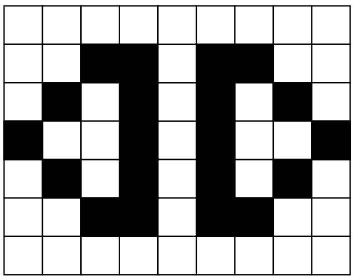
R Pentonimo



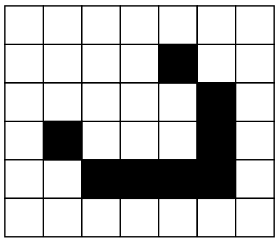
Beacon



Tumbler



Glider2



Spaceship

8 Intrinsic Functions and the HPF Library

8.1 Introduction

Fortran 90 provides the programmer with a rich set of intrinsic functions and subroutines. HPF continues this trend of providing widely used functions and subroutines, and broadens the functionality by providing new functions enabling the programmer to inquire about data mapping: both distribution and alignment.

The reason for providing such a set of functions and subroutines is to allow the programmer to write data parallel algorithms more simply and efficiently, at a higher level and which may also be less prone to bugs. This is done by allowing the compiler to free the programmer from the details of low level implementation, and opens up the possibility of the use of the manufacturer's efficient implementation, tuned for a particular machine architecture.

In particular, the **`HPF_LIBRARY`** module is a collection of procedures containing mapping inquiry subroutines and a set of data parallel primitives such as new reduction operations, combining scatter operations etc. However, the **`HPF_LIBRARY`** module is not a part of subset HPF. The advantage of creating a set of intrinsic procedures (functions and subroutines), is that the interface of these procedures is explicitly known to the compiler, allowing greater syntax checking at compile time.

In this chapter, we shall introduce some of the important groups of procedures and illustrate their use with suitable examples. The optional arguments in the outline of the following functions are shown by "**`[]`**".

8.2 System Inquiry Functions

There are two new inquiry functions in HPF, which query the physical machine. These are **`NUMBER_OF_PROCESSORS`**, which returns the total number of processors available to the program (or number of processors available along a specified dimension of the processor array) and **`PROCESSORS_SHAPE()`**, which returns the shape of the (implementation dependent) processor array (shown in Table 2). These are related to the actual processor array, and should not be confused with the **`PROCESSORS`** directive that may occur in a program.

Table 2: System Inquiry Functions

FUNCTION	VALUE RETURNED
`NUMBER_OF_PROCESSORS([DIM])` integer **`DIM`**	Number of executing processors. Returns an integer.

Table 2: System Inquiry Functions

FUNCTION	VALUE RETURNED
`PROCESSORS_SHAPE()`	Shape of executing processor array. Returns an integer array

Example: For a computer with 8192 processors arranged in a 128 X 64 rectangular grid, the value of **`NUMBER_OF_PROCESSORS()`** is 8192. The value of **`NUMBER_OF_PROCESSORS(DIM=2)`** is 64. For the same array, the value of **`PROCESSORS_SHAPE()`** is (128,64).

Also, the function values remain unchanged throughout the execution of the code, so can be used in the declaration of arrays. For example, declaring,

```
REAL, DIMENSION(3,NUMBER_OF_PROCESSORS()) :: A
```

as a size dependent array allows for an even mapping of the data to processors.

8.3 Mapping Inquiry Functions

HPF provides a number of directives which allow the programmer control over the mapping of data over a set of processors. The mapping inquiry functions, shown in Table 3, are useful for describing the distribution and alignment of an array at run time. This happens with the **`REALIGN`** or **`REDISTRIBUTE`** directives, which need to know which mapping is in effect before attempting to change it. Also, it could be used to make a choice of which algorithm is most appropriate for the current mapping of the data, for efficiency and load-balancing reasons. These subroutines have many arguments, details of which are available in the documentation of these libraries.

Table 3: Mapping Inquiry Subroutines

Subroutine	Effect
`HPF_ALIGNMENT (....)`	Returns information about the alignment of an array.
`HPF_TEMPLATE(...)`	Returns information about the template array to which an array is ultimately aligned.
`HPF_DISTRIBUTION(...)`	Returns information about the distribution of a template or array to which an array is ultimately aligned.

8.4 Computational Functions

8.4.1 Array Location Functions

HPF generalises the Fortran 90 **MAXLOC** and **MINLOC** intrinsic functions with an optional **DIM** parameter for finding the locations of maximum and minimum elements along the specified dimension. Table 4 summarises these functions.

Table 4: Array location functions

Function	Value Returned
MAXLOC(ARRAY[,DIM][,MASK]) type **ARRAY** (not scalar) integer **DIM** logical **MASK**	Location of a maximum value in an array. returns an integer
MINLOC(ARRAY[,DIM][,MASK]) type **ARRAY** (not scalar) integer **DIM** logical **MASK**	Location of a minimum value of an array. returns an integer

Suppose we have a two dimensional array **A** which is distributed **(*,BLOCK(1))** on the processor array **P** as indicated in the figure. To find the location of the minimum value in each column we can use the statement:

```
M = MINLOC(A,DIM=1).
```

For efficiency reasons **M** should be aligned with the columns of **A**.

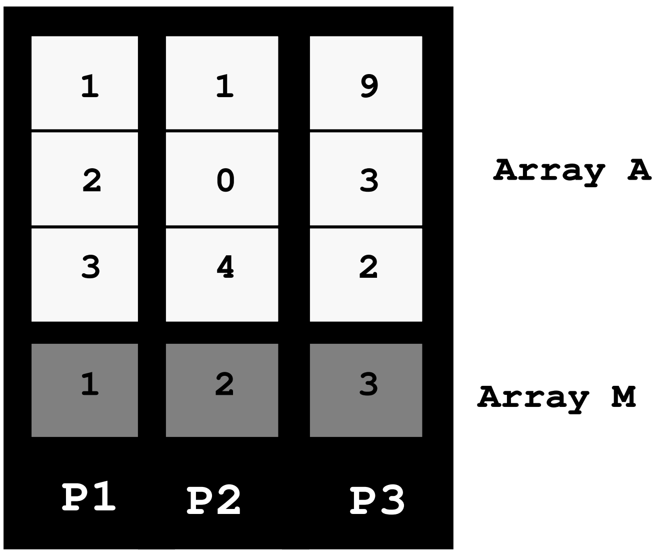


Figure 32: Application of the MINLOC function

8.4.2 Bit Manipulation Functions

The bit manipulation functions of Fortran 90 are extended by the addition of four new functions **ILEN**, **LEADZ**, **POPCNT** and **POPPAR**. These are summarised in Table 5.

Table 5: Bit manipulation Functions

Function	Value returned
ILEN(I) integer **I**	Number of bits required to store an integer. Returns an integer.
LEADZ(I) integer **I**	Number of leading zero bits in an integer's representation. Returns an integer.
POPCNT(I) integer **I**	Number of one bits in an integer representation. Returns an integer.
POPPAR(I) integer **I**	Parity of an integer representation. (1 if odd number of bits set, 0 if even number of bits set.) Returns an integer.

For example, with **I=7**, with binary representation **0111** on a 4 bit machine. **ILEN(I)=4, LEADZ(I)=1, POPCNT(I)=3, POPPAR(I)=1.** A practical use for this is rounding to the nearest power of 2; **2**ILEN(I-1)** rounds up, **2**(ILEN(I)-1)** rounds down.

8.4.3 Array Reduction Functions

The array reduction functions typically combine elements from an array to produce a result of lower rank. For example, in Fortran 90, the **MAXVAL** function returns the maximum value in an array. HPF also provides reduction functions that correspond to binary operations. These functions are summarised in Table 6.

Table 6: Array reduction functions

Function	Value Returned
IALL(ARRAY [,DIM] [,MASK]) integer **ARRAY** (not a scalar), **DIM** logical **MASK** (conformable with **ARRAY**)	Bitwise logical AND reduction Returned value integer (scalar or array)
IANY(ARRAY [,DIM] [,MASK]) integer **ARRAY** (not a scalar), **DIM** logical **MASK** (conformable with **ARRAY**)	Bitwise logical OR reduction. Returned value, integer (scalar or array).
IPARITY(ARRAY [,DIM] [,MASK]) integer **ARRAY** (not a scalar), **DIM** logical **MASK** (conformable with **ARRAY**)	Bitwise logical EOR reduction Returned value, integer (scalar or array).
PARITY(MASK [,DIM]) integer **DIM** logical **MASK** (not a scalar)	Logical EOR reduction. Returned value, logical (scalar or array)

For example, suppose we have three numbers: **ARRAY=(/7, 3, 10/)**.

The binary representations of these numbers are **0111**, **0001** and **1100** respectively.

- **IALL(ARRAY)** is **0010** which is binary for **2**.
- **IANY(ARRAY)** is **1111** which is binary for **15**.
- **IPARITY(ARRAY)** is **1010** which is binary for **10**.

Suppose **MASK** = $\begin{bmatrix} \mathtt{T} & \mathtt{T} & \mathtt{F} \\ \mathtt{T} & \mathtt{T} & \mathtt{T} \end{bmatrix}$. Then **PARITY(MASK,DIM=1)** is $\begin{bmatrix} \mathtt{F} & \mathtt{F} & \mathtt{T} \end{bmatrix}$, and **PARITY(MASK,DIM=2)** is $\begin{bmatrix} \mathtt{F} & \mathtt{T} \end{bmatrix}$.

8.4.4 Array Combining Scatter Functions

HPF allows the use of vector-valued subscripts of arrays (part of Fortran 90) for array assignment. Suppose, we have arrays **A** with value **[10 20 30]** and **B** with value **[3 2 1]**, then the assignment

```
C(B)= A
```

assigns **[30 20 10]** to **C**.

Considering each element, we have

- **C(B(1)) = C(3) = A(1) = 30,**
- **C(B(2)) = C(2) = A(2) = 20,**
- **C(B(3)) = C(1) = A(3) = 10**.

In general, **A** is the source array from which we want to make an assignment, **B** specifies the indices of **C** labelling the elements of **C** to which the assignment is made and **C** is the result array to which the assignments are made. Then for an index **I**, we have

```
FORALL (I = 1:3) C(B(I)) = A(I)
```

In other words, the mapping of values of **A** onto values of **C** is determined by the index array **B**. This mapping, defined using array **B** as a vector subscript, is depicted in Figure 33.

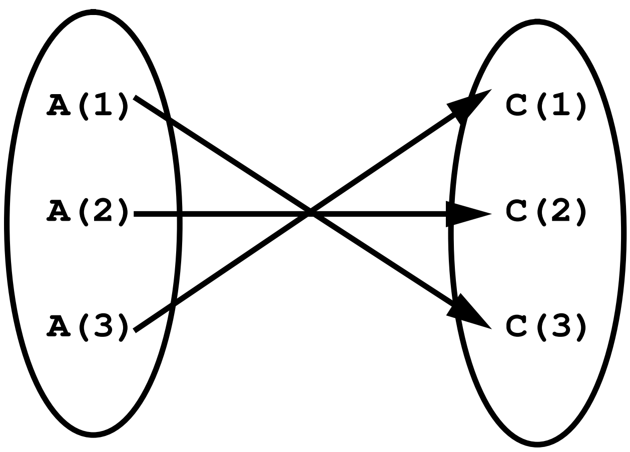


Figure 33: Vector subscript example

However, vector subscripts are useful only to a limited extent. For example, the extension of this operation to subscript arrays with more than one dimension is not defined. That is the index value array, (in the previous example, the array **B**) can not be of more than one dimension. Perhaps more importantly, the vector subscript array cannot contain repeated values on the left hand side of an assignment. If this is the case, then the operation attempt multiple writes to the same atomic data object, which is non conforming in HPF (or Fortran 90). An example of this would be if the array **B** contained **[3 2 3]**, array element **C(3)** would be written twice causing an ambiguity in the assignment.

Often in parallel codes, the programmer wishes to perform some operation which combines generalised data objects in some way. For example, in finite element codes, combining data objects at node points or for some sparse matrix operation, where the spare matrix elements are labelled by some address index. In order to implement such a many-to-many mapping from elements of the arrays, the vector subscript operation is not adequate. A possibility is to use a loop statement, however this is often a very inefficient way of doing this type of operation.

In HPF the vector subscript assignment is generalised to allow array elements to be combined in a completely general way, specified by the programmer by use of index type arrays. These many-to-many mappings are defined by using the combine scatter functions, the idea being that the function combines array elements which are scattered across the arrays. These functions have the form:

```
XXX_SCATTER(ARRAY, BASE, INDX1,..., INDXn [,MASK])
```

where the allowed values for **XXX** are **ALL**, **ANY**, **COPY**, **COUNT**, **IALL**, **IANY**, **IPARITY**, **MAXVAL**, **MINVAL**, **PARITY**, **PRODUCT** and **SUM**, describing the operations applied in combining the array elements.

Each of the arguments are defined below.

- **ARRAY** is the source array from which the elements are mapped. It can be of any type, but must not be a scalar.
- **BASE** is the array whose elements are 'copied' directly to the result. It must be of the same type as **ARRAY**, but need not be conformable with **ARRAY**.
- **INDX1,..., INDXn** are index arguments. These are integer arrays which must be conformable with **ARRAY**. The number of **INDX** array arguments must be the same as the rank of **BASE**.
- **MASK** specifies which elements of **ARRAY** are included in the combine scatter operation. Only the elements of **ARRAY** for which **MASK** is true are included in the operation. This is an optional argument, and must be a logical array conformable with **ARRAY**.
- The result of this function call must have the same type and shape as **BASE**.

In an attempt to make this clearer, consider our previous example, but now implementing it with a combine scatter function. Taking **ARRAY** as **[10 20 30]**, **BASE** as **[0 0 0]** and we use **INDX1** with **[3 2 1]** (**BASE** has rank 1) to define the mapping.

The **RESULT** array, which must be of the same type as **BASE** array, is calculated using the following call.

```
RESULT = SUM_SCATTER(ARRAY, BASE, INDX1)
```

The function is evaluated as follows.

- **RESULT(1) = BASE(1) + ARRAY(INDX1(1)) = 0 + ARRAY(3) = 30**
- **RESULT(2) = BASE(2) + ARRAY(INDX1(2)) = 0 + ARRAY(2) = 20**

- **`RESULT(3) = BASE(1) + ARRAY(INDX1(3)) = 0 + ARRAY(1) = 10`**

To see the full power of the combine scatter function, we can extend this operation to the multidimensional case, in which the array whose elements are to be scattered is of the rank greater than or equal to 2.

For example, if **`ARRAY`** is a **`3*3`** array, then the operation of **`SUM_SCATTER`** can be explained with the help of Figure 34, which shows five arrays, **`ARRAY`**, **`BASE`**, **`INDX1`**, **`INDX2`**, and **`RESULT`**. Both **`INDX1`** and **`INDX2`** are **`3*3`** arrays, required to conform with **`ARRAY`**. **`BASE`** and **`RESULT`** arrays are required to be the same type and shape as each other and the same type as **`ARRAY`**, though no restriction is placed on the extent of these arrays. The **`BASE`** is first copied to the **`RESULT`** array. Then the destinations of elements of the array **`ARRAY,`** to be combined with that element of **`RESULT`**, are chosen according to the indices specified in the arrays **`INDX1`** and **`INDX2`**. In particular, for any pair **`(I,J)`**, the array element **`ARRAY(I,J)`** contributes to the element of **`RESULT(INDX1(I,J), INDX2(I,J))`**.

The procedure for the combine scatter operation can be summarised by the following steps:

- Transfer the values from the **`BASE`** array to the **`RESULT`** array.
- Read a pair of values from **`INDX1`** and **`INDX2`** (**`INDX1(I,J)`** and **`INDX2(I,J)`** in this example).
- This pair indicates the position where the array element **`ARRAY(I,J)`** is mapped to in the **`RESULT`** array.
- Combine the elements **`ARRAY(I,J)`** with **`RESULT(INDX1(I,J),INDX2(I,J))`**, using the appropriate operation, sum in this case. In summary,

  `RESULT(INDX1(I,J),INDX2(I,J)) =`

  `BASE(INDX1(I,J),INDX2(I,J))` ⊕ **`ARRAY(I,J)`**

  where ⊕ represents some combination of elements.

A typical operation is illustrated in Figure 34. A copy of **`BASE`** in transferred onto **`RESULT`**. Now, suppose we do the combine scatter operation for **`I=3`** and **`J=1`**. Reading the values in the cell of **`INDX1(I=3,J=1)`** and **`INDX2(I=3,J=1)`** indicates the destination in **`RESULT`**. From these index values we have a co-ordinate **`(1,1)`** to be updated in **`RESULT`**. The value in **`ARRAY(I=3,J=1)`**, is added to **`RESULT(1,1)`** giving a value of **`4`**. The calculations are repeated similarly for all the remaining cells. In our example, only **ARRAY(3,1)** contributes to **RESULT(1,1)**, but this single contribution to the result need not be the case in general.

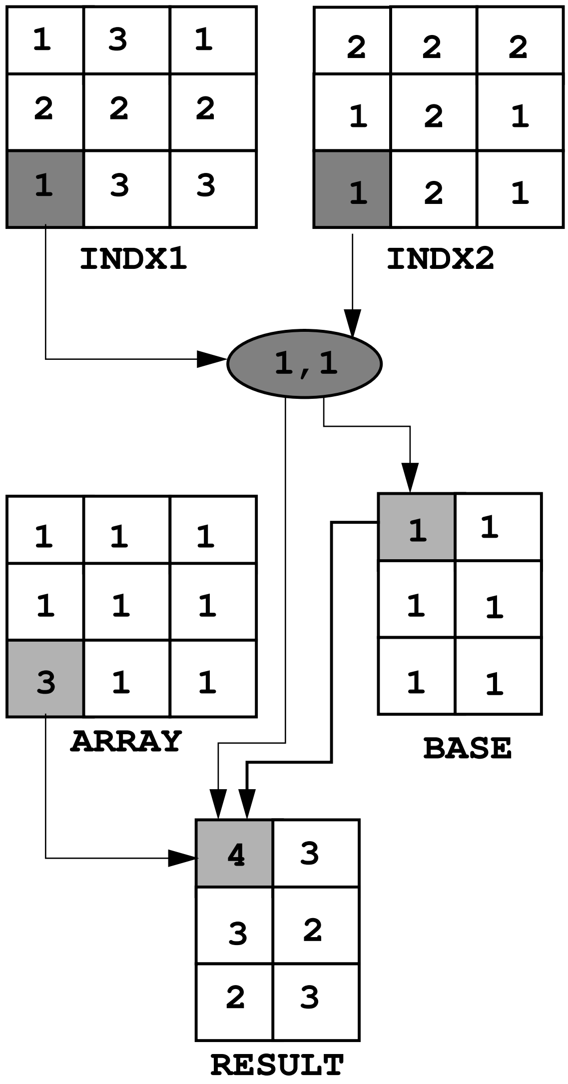


Figure 34: SUM_SCATTER example

8.4.5 Array Prefix and Suffix Functions

As was mentioned in the introduction to data parallel programming, prefix/suffix functions are very important for building parallel algorithms and can be implemented efficiently on parallel machines. For this reason they are included in HPF as library functions.

In a prefix function, or scan of a vector, the value at a given position is calculated by application of some operation on all preceding positions in the vector. Similarly for a suffix function the value at a given position depends on succeeding elements. The direction in which the operation is accumulated is reversed going from a prefix to a suffix function. For example, if we have an array **A** with the value **[1 3 5]**, then **SUM_PREFIX(A)** is **[1 4 9]** and **SUM_SUFFIX(A)** is **[9 4 1]**. The general functions have the form,

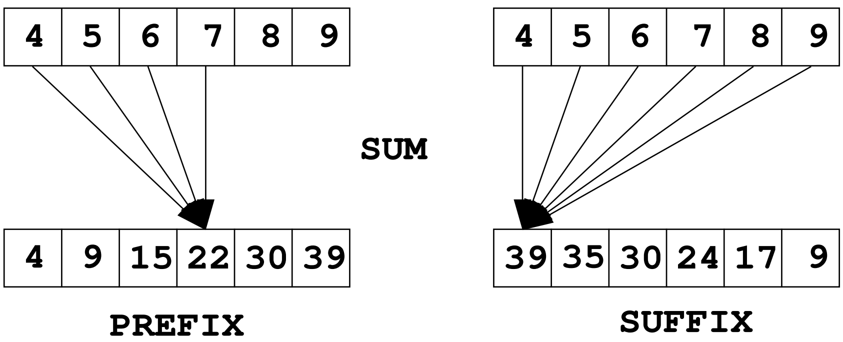
```
XXX_PREFIX(ARRAY [,DIM] [,MASK] [,SEGMENT] [,EXCLUSIVE])
```

The allowed values of **XXX** are **ALL**, **ANY**, **COPY**, **COUNT**, **IALL**, **IANY**, **IPARITY**, **MAXVAL**, **MINVAL**, **PARITY**, **PRODUCT** and **SUM**. Each of the arguments to the function argument is explained below (though there are slight differences for some functions. The HPF specification should be consulted for full details),

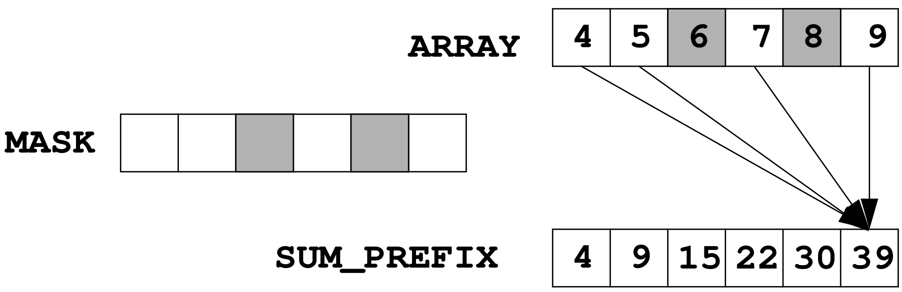
- **ARRAY** is an array of any type, though must not be a scalar
- **DIM** is a integer scalar which specifies the dimension of **ARRAY** along which the operation is performed.
- **MASK** is a logical array, which must have the same shape as **ARRAY**. This defines the elements of **ARRAY** which contribute to the scanning operation. If it is not included all elements of **ARRAY** are considered in the operation.
- **SEGMENT** is a logical array of the same shape as **ARRAY**. This allows the operation to be applied to sections of **ARRAY**, these subarrays are then scanned individually. **SEGMENT** defines a new section by the change from **.TRUE.** to **.FALSE.** or visa versa.
- **EXCLUSIVE** is a logical scalar, defining whether the element of **ARRAY** at that position contributes to the prefix/suffix operation for that position. If it is set **.TRUE.** the element does not contribute to the result.
- the result of this function is an array, of the same shape as **ARRAY**.

Consider the following examples.

1. Suppose, we have an array **ARRAY** which is **[4 5 6 7 8 9].**
   - **SUM_PREFIX(ARRAY)** is **[4 9 15 22 30 39].**
   - **SUM_SUFFIX(ARRAY)** is **[39 35 30 24 17 9].**



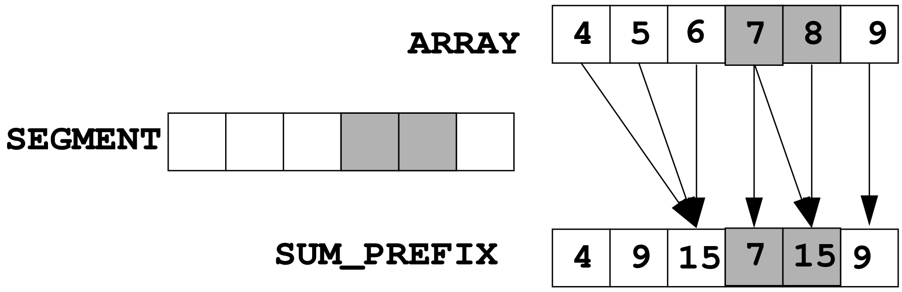
2. With **M** with value **[T T F T F T],**
   - **SUM_PREFIX(ARRAY, MASK = M)** has value **[4 9 9 16 16 25**].



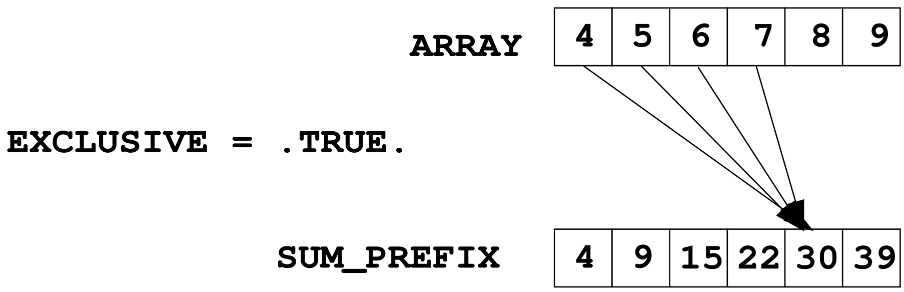
3. With **S** with value **[T T T F F T],** we create three subarrays

ARRAY(1:3), ARRAY(4:5) ARRAY(6) all of which are summed separately.

- **SUM_PREFIX(ARRAY, SEGMENT = S)** is **[4 9 15 7 15 9].**



4. Setting **EXCLUSIVE** equal to **.TRUE.**,
   - **SUM_PREFACE(ARRAY, EXCLUSIVE = .TRUE.)** has value **[0 4 9 15 22 30]**, here, the use of **EXCLUSIVE** means that the first element in the scan does not include itself in the operation, so its value is **0**. However , for the second element we must include everything that comes before it, i.e. **4**.



   - **SUM_PREFACE(ARRAY, SEGMENT = S, EXCLUSIVE = .TRUE.)** has value **[0 4 9 0 7 0].**

5. Consider **ARRAY** $\begin{bmatrix} 10 & 15 & 1 & 4 \\ 3 & 7 & 1 & 23 \end{bmatrix}$, with **M** as $\begin{bmatrix} T & T & F & T \\ F & F & T & T \end{bmatrix}$ and **S** with value $\begin{bmatrix} T & T & F & T \\ T & F & T & T \end{bmatrix}$ (again three subarrays). Then
   - **SUM_PREFIX(ARRAY)** is $\begin{bmatrix} 10 & 28 & 36 & 41 \\ 13 & 35 & 37 & 64 \end{bmatrix}$,
   - **SUM_SUFFIX(ARRAY)** is $\begin{bmatrix} 64 & 51 & 29 & 27 \\ 54 & 36 & 28 & 23 \end{bmatrix}$,
   - **SUM_PREFIX(ARRAY, DIM=2)** is $\begin{bmatrix} 10 & 25 & 26 & 30 \\ 3 & 10 & 11 & 34 \end{bmatrix}$,
   - **SUM_PREFIX(ARRAY, MASK = M)** is $\begin{bmatrix} 10 & 25 & 25 & 30 \\ 10 & 25 & 26 & 43 \end{bmatrix}$,
   - **SUM_PREFIX(ARRAY, MASK = M, SEGMENT = S)** is $\begin{bmatrix} 10 & 25 & 0 & 5 \\ 10 & 0 & 1 & 28 \end{bmatrix}$.

8.4.6 Array Sorting Operations

HPF includes procedures useful for sorting multidimensional arrays. These procedures are structured as functions that return sorting permutations labelling the array elements according to size. Arrays can be sorted along one axis, or sorted by the whole array.

There are two case, **GRADE_DOWN**, in which the permutation labels elements in descending order, and **GRADE_UP**, which produces the indices of array elements in ascending order. They have the following form,

```
GRADE_DOWN(ARRAY [,DIM])
GRADE_UP(ARRAY [,DIM])
```

where **ARRAY** is an array of any type and **DIM** is an integer scalar which specifies the dimension along **ARRAY** which the sorting is done.

The shape of the result depends on whether **DIM** has been specified. If it has, the result is an integer array, the same shape as **ARRAY**. If not, the result is an array of size rank-of-**ARRAY** by number of elements in **ARRAY**. Or more formally, the result has shape **(/ SIZE(SHAPE(ARRAY)), PRODUCT(SHAPE(ARRAY)) /)**.

It should be noted that **GRADE_DOWN** and **GRADE_UP** do not return the actual sorted array. Instead they contain the permutation of the original array which would give the required ordering. If the sorted array is required, it is necessary to use either a vector subscript assignment or some combine scatter function.

Suppose we have an array **A** is **[2 9 4]**. Then if we call

```
S = GRADE_UP(A)
```

then, the result **S** will contain **[1 3 2]**, indicating the ordering, the lowest element in position**1**, the largest in position **2** of array **A**. If we want the sorted array in **X**, then **X** can be obtained from the array **S** and **A** by,

```
X = A(S)
```

However, the vector prefixes can not be used in the general case of multidimensional arrays, instead the appropriate scatter functions must be used.

Consider now a 3*3 array **A** as indicated in Figure 35.

- If we sort the array without specifying the dimension parameter, then we obtain a result array of shape **(2,9)**, rank **2** with **9** elements. Each column of the sorted permutation array S contains the indices of the elements of A in descending order. For example, the element **A(2,2)** is greater than the element **A(2,1)**, so should be closer to the start of the permutation array.
- If we specify the dimension parameter, also shown in Figure 35, then we will have the elements ranked in a descending order along the first dimension (row) or second (column). The result will be of the same shape as **A**, but now the elements of permutation array **S** contain the index of the elements of **A**, ordered by

size along row or column.

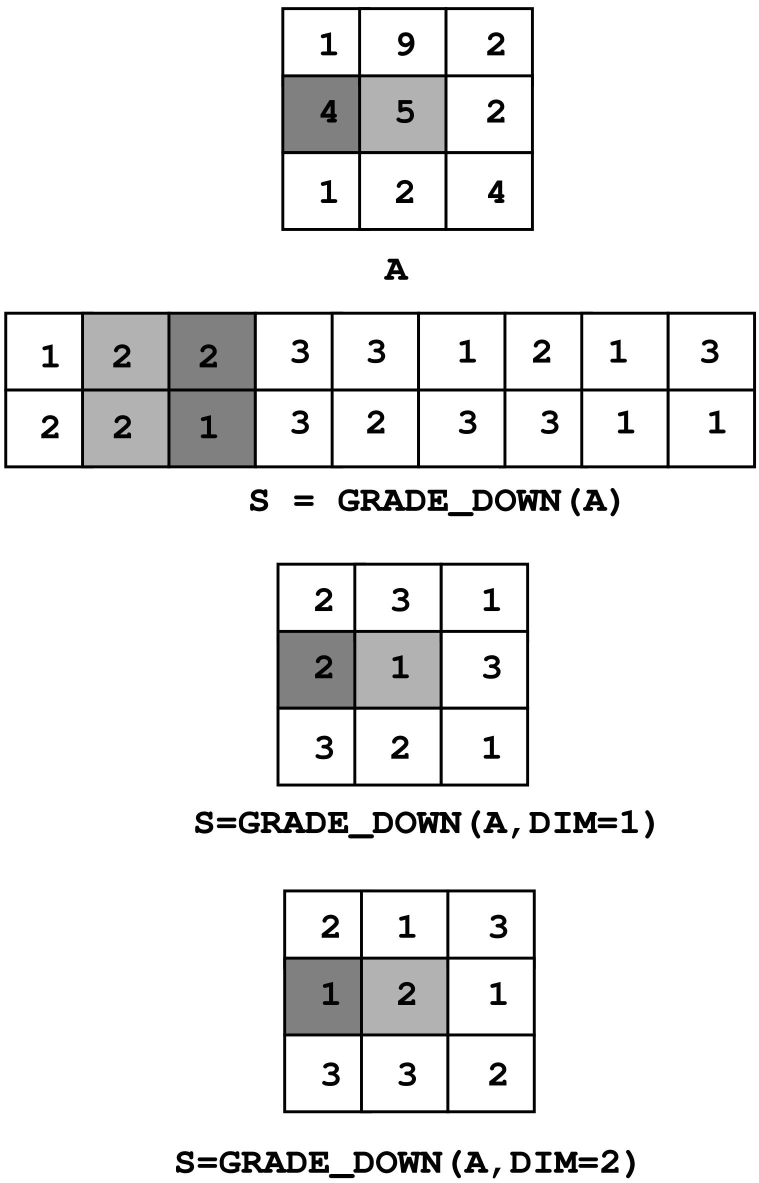


Figure 35: Array sorting operation

8.5 Summary

In this chapter, we have reviewed the set of intrinsic functions and subroutines available in HPF. We also studied a set of functions in the **`HPF_LIBRARY`** module. These functions included array reduction functions, bit manipulation functions, combine scatter functions, prefix and suffix functions and grading functions.

These functions are of importance because they allow the programmer to express data parallel primitives at a high level of abstraction. Also, in many cases, efficient implementations of these functions exploiting particular machine architectures are available. Once quality implementations of HPF become available it would be unlikely that the programmer could code a more efficient general version of these library routines.

8.6 Exercise 7: Golf Scores

The purpose of this exercise is to analyse the following golf score card, using HPF Intrinsic functions.

At each hole there is an “expected” score, say, 3 or 4 shots which is called par. A good golfer should be able to get the ball in the hole in that number of shots.

```
Par    4  4  4  4  4  4   3  4  4  4  3  4  3  5  3  4  5  4
Score  5  3  4  4  4  2   3  5  6  2  5  4  3  4  4  4  7  3
```

The exercise consists of a number of parts - use a different array to store the answers from each part.

- Find the running total scores of the golfer for the round.
- Find the running total scores for the first and second nine holes.
- Find the running totals for Par 3, 4 and 5 holes separately. Use three separate arrays. For each example only set the running total at the holes concerned, set other positions to zero.
- Enumerate the holes on which the player scored a birdie (one less than par). In other words, at the first birdied hole set to 1, then 2 for the second birdied hole and so on. The holes on which birdies were not scored should have value 0.

Use the following program skeleton to base your answer on (an electronic version of this template should be provided for you).

```
PROGRAM golf

! Include HPF library
!
      USE hpf_library

      IMPLICIT NONE

      INTEGER, PARAMETER :: nhole=18

! Declare arrays
!
      INTEGER, DIMENSION(nhole) :: score,par,rtot,rsplit,
     &                             rtot3,rtot4,rtot5,birdie
      LOGICAL, DIMENSION(nhole) :: smask,mask

! Distribute arrays
!
!HPF$ DISTRIBUTE (BLOCK) :: score
!HPF$ ALIGN WITH score :: par,rtot,rsplit,smask,mask,rtot3
!HPF$ ALIGN WITH score :: rtot4, rtot5, birdie
```

```
! Set up score and par
!
      DATA score/5,3,4,4,4,2,3,5,6,2,5,4,3,4,4,4,7,3/,
     &       par/4,4,4,4,4,4,3,4,4,4,3,4,3,5,3,4,5,4/
      INTEGER i

! Initializations
!

!.... 1) Find running total
!
!.... 2) Find running total per 9 holes
!
!.... 3) Find running total for par 3, 4 and 5 holes
!
!.... 4) Enumerate holes where a birdie was scored

       WRITE(*,10) par, score, rtot, rsplit,
     &             rtot3, rtot4, rtot5, birdie

 10   FORMAT(//
     & tr15,' Golf statistics using Scan routines'/
     & tr1,65('-')/
     & '     par: ',18I3/
     & '   score: ',18I3//
     & '    rtot: ',18I3/
     & '  sprtot: ',18I3/
     & '   rtot3: ',18I3/
     & '   rtot4: ',18I3/
     & '   rtot5: ',18I3/
     & '  birdie: ',18I3/)

      END
```

9 Advanced Topics

In this section we describe advanced topics not covered in the main chapters. In particular we consider issues relating to sequence and storage association in Fortran and a procedure interface to external routines written in styles or languages other than HPF.

9.1 Sequence and Storage Association

Fortran was developed in the context of hardware which had a linear memory architecture and there are features in the language that rely on this. For example the array `A(2,3)` is stored in memory in element order `A(1,1), A(2,1), A(1,2), A(2,2), A(1,3)` to `A(2,3)`. The correspondence between two arrays mapped to a linear order is called *sequence association*. The following Fortran 77 fragment illustrates this

```
      PARAMETER (n=100)
      INTEGER A(n,n)
      CALL zero(A,n*n)
      END

      SUBROUTINE zero(A,n)
      INTEGER n, A(n)
      do 1, i = 1, n
1        A(i) = 0
      END
```

and has been used as a standard optimization to zero a multidimensional array. Note that the square array is declared as a 1-D array in the subroutine. The correspondence between elements in the 2-D actual argument and reference by the elements in the 1-D dummy argument is defined by the sequence association. Finally note that the array `A` can be declared as an assumed-size array in the subroutine with the definition

```
INTEGER A(*)
```

An assumed size array is declared with final extent "`*`", this is possible since the compiler does not need to know the extent of the final dimension in order to access the array elements.

A related topic is that of *storage association* which we illustrate with the declaration of two common blocks:

```
      INTEGER A,C
      DOUBLE PRECISION B
      COMMON /rouge/A(10,10),B(50),C(200)
...
      INTEGER P,R
      REAL Q
      COMMON /rouge/P(5,20),Q(50),R(250)
```

The arrays in the **COMMON** blocks share what are called storage units and are associated by storage association. Note that **A** shares storage with **P**, **Q** shares storage with the first half of **B** and also **R** shares storage with the second half of **B** and **C**.

Sequence and storage association poses a problem for parallel implementation in relation to distributed arrays since the linear memory model is no longer valid and excessive remapping would have to occur if it were implemented.

HPF introduces the **SEQUENCE** directive which can be used to inform the compiler that the variable or **COMMON** block should have the sequential property and that sequence and storage association will be supported. In essence, the sequence directive allows for old FORTRAN 77 codes which rely on sequence and storage association to be converted to HPF.

For the subroutine **zero** we can declare `A` to be sequential with

```
!HPF$ SEQUENCE A
```

Similarly we can use

```
!HPF$ SEQUENCE /rouge/
```

to declare that all entities in the **COMMON** block `rouge` are sequential.

To see this more clearly, consider the previous example of sequence association, but now written in HPF with the use of **SEQUENCE** to allow for the linear memory model of FORTRAN 77:

```
        PARAMETER (n=100)
        INTEGER, DIMENSION(n,n) :: A, B, C
!HPF$   SEQUENCE A, B, C ! need HPF$ SEQUENCE if
        CALL zero(A,n*n)  ! array passed by name
        CALL zero(B(1,5),n)! pass an array section
        CALL zero(C(1,10),1)! pass an array element
        END

        SUBROUTINE zero(D,m)
```

```
        INTEGER D(m),m! 1-D array declared
!HPF$   SEQUENCE D
        DO 1, i = 1, m
 1        B(i) = 0
        END
```

In this case we have included the main uses of **SEQUENCE**: when passing an array by its name, passing an array section and passing an array . Note that the **SEQUENCE** directive must be specified for all the arrays in the main program and the dummy array in the procedure called.

The next example outlines the use of sequence in storage association (use with **COMMON** blocks)

```
        SUBROUTINE ENO
!HPF$   SEQUENCE /MADNESS/
        COMMON /GRIDS/ X(100,100),Y(100,100)
        COMMON /MADNESS/MAD(50,50),NESS(10,10)
        END SUBROUTINE

        SUBROUTINE OWT
!HPF$   SEQUENCE /MADNESS/
        COMMON /GRIDS/ X(100,100),Y(100,100)
        COMMON /MADNESS/M(500),ADNE(2,2,2),SS(10,10)
        END SUBROUTINE
```

It is safe to distribute the variables in **GRIDS**, as all variables have same size, type and shape in both subroutines. This would not be the case for the variables in **MADNESS**, however, **!HPF$ SEQUENCE /MADNESS/** guarantees explicit mapping is same in every subroutine.

Note that **EQUIVALENCE** is also used to storage associate objects. Since HPF considers arrays to be mappable by default, care should be taken when porting existing codes since directives may have to be added to make the code run with an HPF compiler.

9.2 Extrinsic Procedures

The designers of HPF were aware that there are those who will wish to program in other languages or styles - for bad or good reasons. Hence HPF allows the programmer to label a procedure as being non-HPF and calls it an extrinsic procedure. An example of the declaration of an extrinsic procedure is

```
INTERFACE
  EXTRINSIC(C) FUNCTION SYSTEM(COMMAND)
    INTEGER :: SYSTEM
    CHARACTER *(*) COMMAND
    INTENT(IN) :: COMMAND
```

```
    END FUNCTION SYSTEM
  END INTERFACE
```

where we define the interface to a C routine. The *extrinsic-kind-keyword* (the text in brackets following the keyword **EXTRINSIC**) defines the type of routine and should this type be known to the compiler it will then be able to build a call to the external routine with the correct arguments (pass by value or reference or possibly converting types). The only extrinsic-kind-keywords defined by HPF are HPF and **HPF_LOCAL**, the former being merely the interface to standard HPF routines. The latter deserves some mention since if implemented it defines the interface to an SPMD execution model where a single program executes on each processor. Hence if an **EXTRINSIC(HPF_LOCAL)** routine were called it would execute independently on each node and have access to data mapped to that node through the arguments. The nodal code could conceivably cause interaction between processors by utilizing a message passing library. On return from the local routine the processors would synchronize. Note that there are restrictions on what the extrinsic routine can do and that an HPF local routine library is described in the HPF standard which could support such a model if implemented.

10 Current Developments

This course has been based on the work of the High Performance Fortran Forum and the standard produced from their work, High Performance fortran Language Specification v 1.0. In January 1994 a meeting was held in Houston, Texas to formally initiate a High Performance Fortran Forum II effort. This was followed by further meetings in August and October. The stated charter for HPFF'94 is:

1. CCI - Corrections, Clarifications, Interpretations including clarifying the ways that HPF 1.0 already supports coarse grained parallelism - e.g. independent loops.
2. Encourage "Industrial Strength" implementations:
   - More notes to implementors
   - Collect and publish practical programming kernels
   - Support establishment of validation suites.
3. Identify, flesh out requirements for HPF 2.0 but do not "develop" these requirements yet. Candidates are:
   - enhanced mapping features
   - parallel I/O,
   - computation control and task parallelism.
4. Produce HPF 1.X document (HPF v1.1 November 1994)

We now briefly consider some of the work currently in progress by the HPFF subgroups, with more detail contained in the HPF-2 Scope of Activities and Motivating Applications report.

10.1 Corrections, Clarifications and Interpretations

The HPF standard was produced in a relatively short time and hence a number of errors and ambiguities still remain. The CCI subgroup has been working to define corrections to the HPF standard, to clarify the points which are not explicitly covered and provide explicit interpretations of the standard. The intention is to provide a CCI document and make it available on the HPFF Web pages.

10.2 Irregular/Task Parallel Benchmarking

This group have been reviewing a number of applications that are useful in motivating the irregular and task parallel extensions to HPF. Another aim is to provide a set of mini applications for which it would be expected that an HPF implementation will produce optimised code.

10.3 Implementations Working Group

This group keeps track of the implementation status of HPF and are thinking of compiling a compendium of papers and essays. The hope is that implementors will share experience so that quality implementations are quickly available.

10.4 Tasking Requirements Working Group

HPF does not address tasking issues which could allow the programmer more control over the more coarse-grained parallelism of applications. Examples of these include computation mapping (allowing the programmer the same control over the distribution of work across a set of processors as is provided for data distribution), reductions and atomic operations (using shared variables to allow iterations of an **`INDPENDENT DO`** loop to affect each other), **`DO ACROSS`** loops (which allows the execution of loops which are prevented from being **`INDEPENDENT`** by loop dependencies, but which can be carried out after some synchronisation condition), allowing multiple HPF processes (generalising the **`EXTRINSIC`** directive and providing means for these separate processes to interact) and parallel section definitions (to allow multiple processes on a single namespace). This group is considering how tasking directives could be combined with HPF data distribution.

10.5 Irregular Data Working Group

The main criticism of data parallel programming and of HPF has been that there is little support for irregular data distribution. Also, there is a need to define more general data alignments and distributions which could deal with unbalanced loads, unstructured data access, dynamically-built data structure. The enhanced mapping features under consideration for HPF-2 include irregular mapping of arrays (allowing arbitrary mapping of data to processors), mapping of linked data structures and derived data and mapping to processor subsets. These are under consideration in this group.

10.6 Parallel I/O Working Group

Parallel input-output facilities in HPF would be desirable to allow the scalable access to data storage. The types of operations of interest include parallel access to files, check-pointing and restart facilities and the ability to overlap computation with I/O. Significant work on parallel I/O was done under HPFF1.0 but was not included in the standard. Recent work has concentrated on the impact of parallel I/O on Fortran standards, and a number of recommendations made to X3J3.

10.7 Information Sources

There is a wealth of information available about HPF, mostly in the electronic form via the Internet. The HPF standard was published in a number of forms. The standard is available online in electronic form by FTP from the host **titan.rice.edu** in the directory public/HPFF. On the World Wide Web there is a server containing information on the HPFF and this is available at the URL **http://www.erc.msstate.edu/hpff/home.html**.

There is also a mailing list for those interested in keeping up to date with HPFF developments and discussions. Send the message "add hpff" to the address **hpff-request@cs.rice.edu**. In particular note that this mailing list receives postings of minutes of HPFF meetings.

11 Course Exercises

11.1 Exercise 1: Introduction to the Compiler

Write a Fortran 90 code to evaluate the discontinuous function:

$$0 \le x < 400: \quad y = x * x * x + C1$$

$$400 \le x < 1000: \quad y = x * x + C2$$

$$1000 \le x \le 10000: \quad y = x$$

- **Declarations**:

  Declare two vector integer arrays, **`x`** and **`y`**, each with 10000 elements, and two integer scalars, **`C1`** and **`C2`**.

- **Initialisation**:

  Initialise the vector **`x`** to be

  ```
  x(i) = i, for all values of i from 1 to 10000
  ```

  using the vector subscript array assignment of Fortran 90.

- **Calculation**:

  Assign the corresponding elements elements of y. Use either

  - the array subscript notation to assign array sections
  - or use the **`WHERE`** statments on the whole array ,

  Print out your answers for *y(50)*, *y(500)* and *y(1500)*. Check that your code gives the correct results (125005, 250010 and 1500 respectively).

- **Compilation**:

  Call your program **`ex1.hpf`**, and compile it using the script provided:

  ```
  hpf -compile ex1
  ```

  This converts the code into FORTRAN 77 with calls to a communication library and then produces an executable.

- **Running on a sinlge node**:

  Run the executable on a single processor, using the **`-run`** option for a single processor (check you answers agree with the ones above):

  ```
  hpf -run ex1
  ```

- **Profiling:**

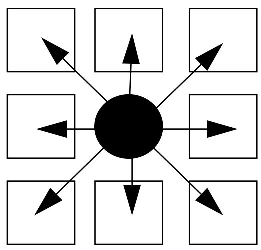
  Profile the code by including the **`-profile`** option on everythe command line. Remember to comment out all the input/output in the original code. Examine the output produced.

11.2 Exercise 2: The Game of Life

The aim of this exercise is to show how Fortran 90 can be used to program the Game of Life, a simple grid based problem with complex behaviour. It will show how Fortran 90 can be used to produce code in a very neat form and exposes the potential for coding in a data parallel programming style.

11.2.1 The Game of Life

The game of life is a simple cellular automata where the world is a 2D grid of cells which have two states: alive or dead. At each iteration the new state of a cell is determined by the state of its neighbours at the previous iteration. This includes both the nearest neighbours and diagonal neighbours, i.e.



The rules for the evolution of the system are;

- if a cell has exactly two alive neighbours it maintains state.
- if it has exactly three alive neighbours it is alive.
- otherwise, it is dead.

Your code will need to

1. Initialise board

Start loop

2. Print board
3. Calculate number of neighbours
4. if (neighbours = 3) then live
if (neighbours < 2) or (neighbours > 3) then die

End loop

The number of neighbours can be calculated using shifts, e.g.,

```
target = CSHIFT(source, shift, dimension)
```

sets **target** to be the same as **source** but with its elements shifted a distance **shift** along dimension of the array **dimension**. For example,

```
target = CSHIFT(source, -1, 1)
```

would set

```
target(i) = source(i - 1)
```

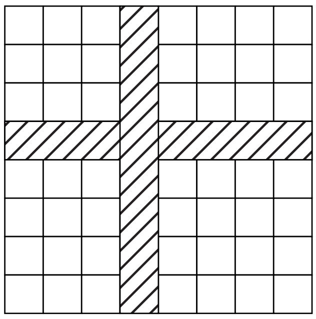
CSHIFT automatically performs periodic boundary conditions. Otherwise references would be made to elements outside the bounds of the array.

11.2.2 The Exercise

A template for this exercise is provided, which includes pointers to the completion of this exercise and also all the print statements necessary for viewing the results.

- **Initialisation**:

  Use array syntax to initialise a board (an **N×N INTEGER** array with value 1 for a live cell and 0 for a dead one) with the following pattern, where shaded cells are alive, where **N = 8** in this case. Set the row and column nearest the centre (i.e. **N/2**)



- **Print board:**

  The board can be printed either as a plain text file using the standard Fortran print format statements, or using the following pieces of code can be included to produce a series of bitmaps which can be animated using **xv**.

```
!      Include in the declarations; strings for filenaming
       CHARACTER (LEN=10) :: picfile
!...
! Include in the time loop
! hack to write time into a string
       WRITE(picfile,fmt='(''life'',I2.2,''.pgm'')') count

       OPEN(UNIT=10,FILE=picfile)
```

```
! write the board to a pgm file for viewing
      WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 1
! must be a capital P for the MAGIC NUMBER
      WRITE(10,*) board
       CLOSE(10)
```

This should, when run, produce **life_*.pgm** files which can be viewed using **xv**. Alternativly, these files can be viewed as animation using,

```
xv -expand 10 -wait 0.5 -wloop -raw *.pgm
```

- **Update**:

  Declare an array the same size as the board (**N**×**N**)to contain the number of neighbours for each point. Include all nearest neighbours, including diagonal neighbours. This update can be done using the following Fortran 90 features.
  - Use **CSHIFT** to calculate the number of neighbours. In order to access the diagonal neighbours you may need to used nested **CSHIFT**s. For example,

    ```
    target = CSHIFT(CSHIFT(source, 1, 2), -1, 1)
    ```

    This would shift array **source** initially in the 2nd dimension and then in the 1st.
  - Use **WHERE** to decide whether to create or kill new organisms at each grid point.
- **Compilation**:

  Compile your code (**life.hpf**, say, board size **N=8** for the test, for 10 iterations of evolution) with the script as in the previous exercise (**hpf -compile life**) to produce an executable **life**.
- **Single node**:

  Run the code on a single processor for 10 iterations to check produce a set of (**hpf -run life**).

  Use **xv** to view the resulting arrays to check your code is working.

11.2.3 Extra Exercise 1

CSHIFT was used to count the number of neighbouring live cells. You can optimise the code to do the count with only four **CSHIFT** operations instead of eight. This requires the use of a temporary array the same size as **board**. For each array element,

- create the temporary by adding the current value of **board** to the value of the left and right partners (two **CSHIFTS**)
- find the upper and lower partners for this temporary array (two more **CSHIFTS**) and add these to the temporary array
- each element now contains the number of neighbours, plus the original value of the cell. Subtract the original state to obtain the number of neighbours with four **CSHIFT**s.

11.2.4 Extra Exercise 2

Try to count and display the number of alive/dead/new/killed cells at each iteration. You will probably need to use some additional Fortran 90 features.

11.2.5 Extra Exercise 3

Define an extra array which is used for display. In this array keep track of which generation (modulo 256) that the cell was born. This array can be used for a colour display and so the age of the cell can be seen. To print out this colour array, change the header statement from

```
WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 1
```

to

```
WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 255
```

11.3 Exercise 3: Life Distributed

In this exercise we return to Exercise 2 and rewrite the codes including explicit data distribution directives.

- **Distribute Data**:

  Introduce data mapping directives into the code to distribute the data. So, for example, for 2D arrays **`source`** and **`target`** you could use

  ```
  !HPF$ DISTRIBUTE (BLOCK,BLOCK) :: source, target
  ```

  to distribute the elements of these arrays in a block form.

- **Compilation**:

  Compile the codes as before, using the **`hpf`** script. Increase the size of your array to **`N=100`**.

- **Single Node**:

  Run the executable on a single node. Check it gives the same results as before.

- **Multiple Nodes**:

  First create a **`MYHOSTS`** file in the working directory with four workstation names including your own (the training room machines are called **`young`**, **`sutherland`**, **`stewart`**, **`crawford`**, **`ramsay`**, **`blair`** and **`coventry`**). You need to have a **`MYHOSTS`** file in each directory where you run executables on more than one processor).

  Run the code on four processors, using

```
hpf -run -nodes 4 life
```

and check you still get the same answer!

- **Profiling**: Repeat the last stages with the **-profile** option set in each command. Remember to comment out all the input/output statements when profiling.

Use the resulting information to investigate how the performance of these examples changes with different data distributions. In particular profile the codes and see how the amount of communication changes. What is the most efficient distribution for the Game of Life and why?

For example, change the distribution of the 2D arrays in the Game of Life from **(BLOCK,BLOCK)** to, say, **(CYCLIC,CYCLIC)** or using a degenerate distribution.

11.4 Exercise 4: The Mandelbrot Set

The Mandelbrot Set is the set of numbers resulting from repeated iterations of the following complex function:

$$Z = Z^2 + C$$

which, separating real and imaginary form, looks like,

$$z_i = 2.0 * z_r\, z_i + c_i$$

$$z_r = z_r^{\,2} - z_i^{\,2} + c_r$$

for complex numbers, Z and C. This function is defined for complex values of $C=(c_r,c_i)$ in the range $([0.0,1.0], [0.0,1.0])$, with the initial conditions, $z=c$. In the case study, we will mainly be concerned with the Mandelbrot Set defined in the first quadrant of the complex plane, i.e. we will consider c in the range ([0.0, 1.0], [0.0,1.0]).

What is normally plotted is the number of iterations taken for z to reach some threshold value. We will take this threshold as

$$|z|^2 > 4.0$$

and set an upper iteration limit of 255. This number is converted into a greyscale or colour, and plotted at the point whose coordinates are (c_r , c_i).

11.4.1 FORTRAN 77 Code

Below is a fragment of a serial code which computes the Mandelbrot set. This should outline the algorithm to be used in the full HPF version.

```
c Declare N*N arrays for real and imaginary parts of
c arrays C and Z
c i.e. declare real arrays CR, CI, ZR, and ZIs
c     ...
c Initialise arrays CR, CI
c Initialise arrays ZR=CR, ZI=CI
c     ...
c Initialise ZIS and ZRS to hold the squares of ZR and ZI
c     ...
      DO i = 0, 255
        DO j = 1, N
          DO k = 1, N
            IF (ZRS(j,k) + ZIS(j,k) .LE. 4.0 ) THEN
              ZRS(j,k) = ZR(j,k) * ZR(j,k)
              ZIS(j,k) = ZI(j,k) * ZI(j,k)
              ZI(j,k) = 2.0 * ZR(j,k) * ZI(j,k) + CI(j,k)
              ZR(j,k) = ZRS(j,k) - ZIS(j,k) + CR(j,k)
              COLOUR(j,k) = i
            END IF
          END DO
        END DO
      END DO
c ...
```

11.4.2 HPF Version: Serial

Write an HPF program, based on the above code fragment, which will compute the Mandelbrot set, using the HPF/Fortran 90 array features to carry out the various stages (a template is available).

First off write a serial HPF version. A template is included with various pointers on how to proceed (and also the print statements required for a graphical output). The following steps are needed:

- **Initialisation**

  Initialise the *rows* of **CR** (real component of *C*) to be in the range [0.0,1.0] (i.e. every element in the first row has value 0.0, every element in the last row has value 1.0, and the intermediate rows have values varying between 0.0 and 1.0). Initialise the *columns* of **CI** (imaginary component of *C*) similarly. Use **FORALL** for this initialisation.

  Set the initial conditions **ZR** and **ZI** to be **CR** and **CI** respectively and define variables **ZRS** and **ZIS** to be the square of **ZR** and **ZI** respectively.

- **Iteration**

  Rewrite the above FORTRAN 77 code using Fortran 90 array syntax, using the **WHERE** construct.

  This program takes each point at a time and iterates the complex function (up to a maximum number of iterations, **RESOLUTION = 255**). The iterations stop when the absolute value of Z reaches or exceeds 2. The colour values are a measure of how many iterations it took to “escape” to 2 at each point.

- **Output**

  To write out the **COLOUR** array as a bitmap, use similar instructions as were used in the Game of Life. Include the following lines of code at the end of your program (this should already be in the template),

```
OPEN(UNIT=10,FILE='mandel.pgm')
WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 255
WRITE(10,*) colour
CLOSE(10)
```

Compile and run the code on a single workstation, with the size of the arrays set with **N = 128**. View the resulting bitmap with

```
xv -gamma 7 mandel.pgm
```

and check that the code works correctly (i.e. you recognise the bitmap to be the well known Mandelbrot set).

11.4.3 HPF Version: Parallel

Include the necessary HPF data mapping directives to distribute and align the arrays. Try both **BLOCK** and **CYCLIC** distributions. With this done, all the work is automatically distributed and the code “should” run in parallel.

Compile and run the code for 4 nodes. Check that the answer is still the same.

Use the profiling option to time the execution of the code and the amount of message passing involved.

Use this to compare how the performance is affected by use of different data distributions. Remember to comment out the input/output statements in the code while profiling.

11.5 Exercise 5: Birthday

The purpose of this exercise is to choose appropriate **DISTRIBUTE** and **ALIGN** directives for a particular problem.

Our example is that of finding the day of a year any given individual’s birthday falls within a particular (non-leap) year. A simple way to do this is to produce a look-up table with the number of days in each month (or more appropriately, a running total of these values), which is consulted for every birthday. The day on which the birthday falls is the sum of the total number of days so far plus the number of days in that particular month.

The following code defines data structures and data initialisation along with a print statement useful for viewing the results (a template is provided with code already written).

```
      PROGRAM birthdays
      IMPLICIT NONE
      INTEGER, PARAMETER :: n=1000
      INTEGER, DIMENSION(n,2) :: dob
      INTEGER, DIMENSION(n)   :: days
      REAL, DIMENSION (n) :: rand
      INTEGER i
      INTEGER, DIMENSION(12) :: days_in_month,days_so_far
      DATA days_in_month/31,28,31,30,31,30,31,
     &              31,30,31,30,31/
!     ...
!     Fill in this part
!     ...
      PRINT 1,dob(:15,1),dob(:15,2),days(:15)
 1    FORMAT(' DOB: month : ',15i4/
     &       ' DOB:   day : ',15i4/
     &       '       days : ',15i4/)
      END
```

The array, **days_in_month**, is initialised to contain the number of days in each month. You will also find it useful to produce another array, **days_so_far**, which contains a cumulative total of the number of days in each month.

The array **dob** is used to hold each individual's date of birth with **dob(:,1)** holding the months and **dob(:,2)** the days within the month. Initalise this to contain a random set of birthdays, using the Fortran 90 intrinsic subroutine, **RANDOM_NUMBER**. to create a set of random numbers in the range [0,1] in array **rand**.

Your task is to complete the program and calculate the number of days from the beginning of the year for each birthday. The result should be assigned to the array **days**.

Firstly decide which arrays should be distributed and how. Use a mixture of **DISTRIBUTE** and **ALIGN** directives to do this.

Pay particular attention to the alignment of the array **days_in_month/days_so_far**.

Compare the performance of the code when run on a single and multiple workstations, and also when the distribution statements are included.

11.6 Exercise 6: Life in a subroutine

The aim of this exercise is to rewrite the distributed Game of Life from Exercise 3, with the iterations of the update of the system done in a subroutine.

Before doing this, niclude all the data mapping constructs you have met so far (i.e. set the **PROCESSORS** directive and use **ALIGN** to specify the relationship between the board and its neighbours).

The subroutine should take the board array and the number of iterations as inputs. Within the subroutine, the update should be performed and the result printed to file as before. All iterations of the update should be done in this subroutine.

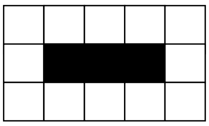
- Start with a descriptive mapping. Copy the distribute directives for the actual arrays in the main program into the subroutine (to distribute the dummy arguments in the same way).
- Generalise to a prescriptive mapping (distribute the actual arguments with a different mapping than the dummy arguments)

In each case, you should include an interface block for each subroutine, in the main program.

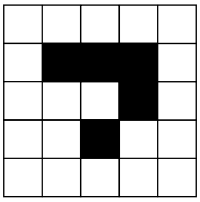
Compare the performance of these two cases, trying to see the overhead introduced by remapping in the prescriptive case.

11.6.1 Extra Exercise

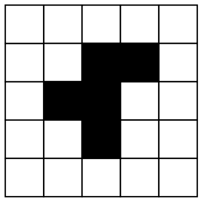
Try some of the following patterns as initial conditions for you board:



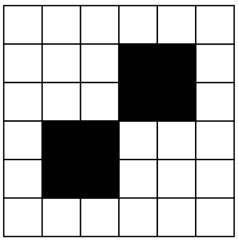
Blinker



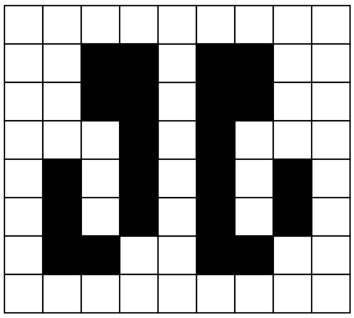
Glider



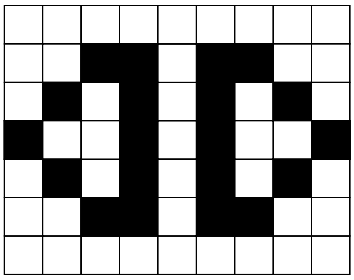
R Pentonimo



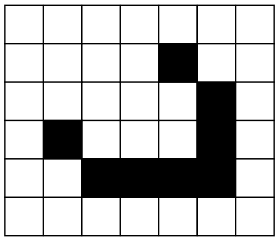
Beacon



Tumbler



Glider2



Spaceship

11.7 Exercise 7: Golf Scores

The purpose of this exercise is to analyse the following golf score card, using HPF Intrinsic functions.

At each hole there is an “expected” score, say, 3 or 4 shots which is called par. A good golfer should be able to get the ball in the hole in that number of shots.

```
Par    4 4 4 4 4 4  3 4 4 4 3 4 3 5 3 4 5 4
Score  5 3 4 4 4 2  3 5 6 2 5 4 3 4 4 4 7 3
```

The exercise consists of a number of parts - use a different array to store the answers from each part.

- Find the running total scores of the golfer for the round.
- Find the running total scores for the first and second nine holes.
- Find the running totals for Par 3, 4 and 5 holes separately. Use three separate arrays. For each example only set the running total at the holes concerned, set oth-

er positions to zero.

- Enumerate the holes on which the player scored a birdie (one less than par). In other words, at the first birdied hole set to 1, then 2 for the second birdied hole and so on. The holes on which birdies were not scored should have value 0.

Use the following program skeleton to base your answer on (an electronic version of this template should be provided for you).

```
PROGRAM golf

! Include HPF library
!
      USE hpf_library

      IMPLICIT NONE

      INTEGER, PARAMETER :: nhole=18

! Declare arrays
!
      INTEGER, DIMENSION(nhole) :: score,par,rtot,rsplit,
     &                             rtot3,rtot4,rtot5,birdie
      LOGICAL, DIMENSION(nhole) :: smask,mask

! Distribute arrays
!
!HPF$ DISTRIBUTE (BLOCK) :: score
!HPF$ ALIGN WITH score :: par,rtot,rsplit,smask,mask,rtot3
!HPF$ ALIGN WITH score :: rtot4, rtot5, birdie

! Set up score and par
!
      DATA score/5,3,4,4,4,2,3,5,6,2,5,4,3,4,4,4,7,3/,
     &       par/4,4,4,4,4,4,3,4,4,4,3,4,3,5,3,4,5,4/
      INTEGER i

! Initializations
!

!.... 1) Find running total
!
!.... 2) Find running total per 9 holes
!
```

```
!.... 3) Find running total for par 3, 4 and 5 holes
!
!.... 4) Enumerate holes where a birdie was scored

      WRITE(*,10) par, score, rtot, rsplit,
     &             rtot3, rtot4, rtot5, birdie

 10   FORMAT(//
     & tr15,' Golf statistics using Scan routines'/
     & tr1,65('-')/
     & `     par: `,18I3/
     & `   score: `,18I3//
     & `    rtot: `,18I3/
     & `  sprtot: `,18I3/
     & `   rtot3: `,18I3/
     & `   rtot4: `,18I3/
     & `   rtot5: `,18I3/
     & `  birdie: `,18I3/)

      END
```