High-Performance Fortran for SPMD Programming: An Applications Overview

Sanjay Ranka, Hon W. Yau, Kenneth A. Hawick, and Geoffrey C. Fox

CRPC-TR97722
June 1997

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted in November 1997

High-Performance Fortran for SPMD programming: An Applications Overview

Sanjay Ranka
Computer and Information Science and Engineering Department
University of Florida
Gainesville, FL 32611–6120
USA
Hon W Yau
Edinburgh Parallel Computing Centre
University of Edinburgh
Edinburgh EH9–3JZ
Scotland/UK
Kenneth A Hawick
Department of Computer Science
The University of Adelaide
South Australia 5005
Australia
Geoffrey C Fox
Northeast Parallel Architectures Center
Syracuse University
Syracuse, NY 13244–4100
USA

June 3, 1997

Abstract

This paper evaluates the High-Performance Fortran (HPF) language as a candidate for implementing scientific, engineering and computer science software on parallel machines. The paper reviews major HPF language features and discusses general algorithmic issues common to broad classes of SPMD applications. It also discusses limitations of the current specification and the extensions required for effective representation and parallelization of several applications.

This paper is also available on the NPAC web site under the URL:

`http://www.npac.syr.edu/hpfa/Papers/HPFforSPMD/`

Contents

List of Figures

List of Tables

---

1 Introduction

From the early 1980's to the mid 1990's, parallel computer architectures went through a proliferation in designs, as manufacturers competed to meet the promise of high performance and/or low hardware costs [41]. The machine from that era typically boasted esoteric hardware features with which to distinguish themselves from the existing vector-parallel supercomputers and other traditional mainframes. This included exotic interconnect topologies (*e.g.*, hypercubes), specialised processing units (*e.g.*, the Inmos Transputer), high-performance proprietary processors (*e.g.*, KSR-1) and large arrays of (also proprietary) bit-processors (*e.g.*, MasPar, TMC CM-1). However, the increasing sophistication and power of commodity CMOS components has resulted in a consolidation towards designs which spread research and development costs through the use of more general-purpose parts. Hence today, the most prevalent parallel computer designs extensively use hardware developed for other markets; currently this is mainly the workstation market, but it is only a matter of time before Intel x86 processors from the personal computing world are also exploited. Moreover, communication libraries are now designed such as to abstract away the underlying topology of the hardware, such that an application programmer should not be expected to worry whether a machine is wired in a 3-D torus or in a hypercube. So from the in terms of the machine's processing units, the *coarse grained* design is now the dominant architecture (see §2), whilst developments in software has now made the question of interconnect topologies an implementation issue. It is because of these software

developments that the question of writing portable parallel applications can be answered, a task for which the High Performance Fortran language was specifically designed to tackle.

Currently, the most prevalent means of writing parallel applications for coarse grained machines is by calling message-passing library routines to pass data amongst the processors. A large number of the codes written in such a manner are *single program, multiple data* (SPMD) in nature, where the same program is executed on each of the participating processors. However, message-passing programming is generally regarded to be tedious and error-prone, as the programmer has to manage the flow of data to each of the processors at each computation step, by hand. Furthermore, portability of the code can be a problem when libraries unique to a particular hardware vendor are used. The acceptance of the machine-independent message passing interface 'MPI' [34] has largely relieved the problem of portability for most message passing codes, but message-passing software development tools are still strangely lacking.

An alternative to hand-coded message-passing is to develop compilers capable of generating automatically the message-passing instructions required. Unfortunately, attempts to develop parallelizing compilers for widely-used languages like 'C' and Fortran have not been successful. Although fully automatic parallelizing compilers can produce excellent results on isolated loop nests, they often fail to exploit available parallelism in realistic application because of the following reasons:

1. The compiler's analysis can simply fail, reporting a dependence when none exists.

2. An actual dependence may be an artifact of a sequential optimization, such as reusing an array's storage to save memory. In these cases, it is often possible to rewrite the program to allow the parallelization to take place, if the user can detect the problem.

3. The program may use an inherently sequential algorithm, or an algorithm with limited parallelism. For example, the standard method of solving a tridiagonal system uses a first-order recurrence that cannot be directly parallelized. In this case, the best option is to change to a different algorithm.

With items 1 and 2, the failure of the compiler to notice the parallelisation in the code is due largely to the inherently serial design of the Fortran programming language. That is to say, given the syntax of the language, the determination of data dependencies at compile time would necessarily be limited, hence forcing the compiler to make possibly pessimistic assumptions. Given the Fortran language, a complete analysis of the parallelism in an arbitrary code will require knowledge only available at run-time.

As a result, efficient use of distributed memory systems requires either the use of message-passing programming or the development of new languages. To address some of the above issues, a specification for a language called High-Performance Fortran (HPF) was adopted in 1993 [18]. The HPF language specification is based in part on experience gained with the research languages such as Fortran D [21, 22, 10] and Vienna Fortran [5]. HPF is based on

Fortran 90 [1, 33], an extension of Fortran 77 that allows new features to be integrated into existing code in a controlled, evolutionary manner. Fortran 90 includes major new features (over Fortran 77) in a broad range of areas, including new control structures, array processing, dynamic memory management, data types and structures, operators, object-orientated features, operator overloading, subprograms, and global data. From the perspective of parallelism, the most significant extension provided by Fortran 90 is the treatment of arrays as individual data objects. This feature is inherently well-suited to data-parallel operations, which are readily processed by almost all parallel machines.

A critical issue for parallel computing is the problem of scalability and portability of the software, since this is the key for protection of software investment. This issue is seriously addressed by HPF with its minimal set of extensions to Fortran 90 (see §3) for supporting the data parallel programming model, which is defined as single-threaded, loosely-synchronous parallel computation using a global name space. One fundamental extension to Fortran 90 provided by HPF concerns the use of compiler directives for specifying the distribution of data among multiple processors. These directives provide information to the compiler on where the parallelism of the problem can be exploited, and yet may be safely ignored by a serial compiler. Indeed, the language only include the '`FORALL`' first class statement/construct for more generalised data-parallel computations, and the '`PURE`' attribute for declaring 'data-local' subprograms: both of which have subsequently been adopted for the Fortran 95 standard (see §3.4).

The Fortran programming language has traditionally been most widely used in the scientific and engineering fields, and in particular in the solution of multidimensional partial differential equations. With these problems, the usual procedure is to first map the problem from continuous to discretised space, via a meshing procedure. The problem can then be represented with state variables at points on this mesh —for example, the fluid pressure value in a computational fluid dynamics simulation (*e.g.*, in §6.6). By representing these states on the grid points as an N dimensional vector (where N is the number of grid points), the solution vector for a given boundary condition can be found through the solution of an $N \times N$ matrix. The key point is that this matrix reflects the topological structure of the underlying mesh, and is hence likely to be sparse, since very few problems have physics where every point in state space depends on the state of every other points. Furthermore, in the case of an *irregular* mesh, where each element may be topologically variant, the matrix will also have an arbitrary sparse structure (see [8] for a more detailed discussion on sparse matrix problems). Such irregular meshes also require a more complicated data structure (since determining neighbouring nodes is non-trivial) involving indirect addressing with gather/scatter operations (see §6.3), with the distinct possibility of load imbalances from the varying sized elements. In contrast, *regular* mesh problems with homogeneous topologies involve matrix-vector & vector-vector operations for direct solution methods (§6.1), or '`CSHIFT()`' operations for iterative solution methods (§6.2), which are easier to implement efficiently on current HPF compilers.

The current HPF language specification supports the data parallel programming for several practical applications, but has limited support for unstructured problems with irregular data mappings. A new round of discussions concerning a second HPF specification ('HPF-2.0') have just been completed [18], and one of the key areas of discussion concerns better support for irregular mesh codes (see [19] for the motivating applications, and §3.5 for a discussion of these new features). It is our belief that HPF will complement —not replace— programming using message-passing libraries. For many applications, HPF is expected to provide convenient, compact and relatively-simple programs without recourse to explicit message-passing instructions. However, other applications are expected to require hand-coded message passing to yield acceptable utilization of distributed-memory systems.

Although HPF and, to a slightly lesser extent Fortran 90, are comparatively new languages, it is encouraging to see that many vendors either support, or is highly interested in supporting HPF, as is seen in documented reports [36, 30, 43] comparing some of these products. This reinforces the wish to have a language portable across a variety of platforms and compilers. In particular [43] has observed scalability up to 8 processors on simple application codes, with modest overheads when running from Fortran 90 to HPF. In these cases, it is difficult to envisage a hand-tuned message passing code providing appreciably better performances. For real-world applications in the commercial world, success has also been reported for a large ocean modelling code [40], where HPF was used in the parallelisation of a multiple tridiagonal matrix solver.

This paper provides a brief overview of High Performance Fortran, general principles of its compilation and runtime support for scalable code generation for coarse grained machines. The main focus is on a detailed analysis of HPF features required for the parallelization of a number of regular SPMD applications.

2 Coarse Grain Parallel Machines

Many of the specifications of the High Performance Fortran language deals with the problem of mapping the elements of a distributed arrays onto a multiprocessor machine. Specifically, a coarse grain, general purpose parallel machine —in contrast to fine-grained specialised machines such as the Illiac-IV, TMC CM-1, 2 & 200 and the AMT DAP, which have their own custom-designed programming languages and associated restrictions. That this is an important architecture for HPF to address can be seen by the comparative rarity of fine grained parallel machines, with respect to their coarse grained counterparts. This in turn is due to the increasing use of commodity components in the construction of high performance computers, which spreads development costs and helps to protect existing software bases.

Coarse grained parallel machines consist of a set of processing elements (typically tens to a few thousand) connected through an interconnection network. The memory on these are either distributed with a fraction local to each processor, or shared through a common interconnect: the former requires interprocessor communication via message passing calls and

the latter more simply via a shared address space. Topologically, the interconnection network of these computers are varied. They can be 2D meshes (Paragon, Delta), 3D meshes (Cray T3D & T3E), hypercubes (nCUBE), fat tree (TMC CM5, Meiko CS-2), hierarchical networks (cedar, DASH), hierarchical rings (KSR-1), a common switch unit (IBM SP-2), or a shared bus (SGI Challenge). Since shared-memory machines are limited by bus to (typically) 32 processors, some machines have amalgamated distributed memory topologies to achieve scalability beyond that limit (SGI/Cray Origin). Finally, one may have a physically distributed memory architecture, but a conceptually shared-memory programming model (KSR-1, HP/Convex Exemplar).

Where the parallel machine has a distributed memory model, its access to local memory are significantly faster than non-local memory, typically by an integral factor. The cost of accessing non local data can be simplistically modeled as $t_s + t_w m$, where t_s represents the handshaking latency, t_w represents the inverse bandwidth of the underlying network, and m the size of the message size.

The startup time t_s is often large, and can be several hundred machine cycles or more. The per-word transfer time t_w is typically an order to two orders of magnitude larger than t_c, the time to do a unit computation on data available in the local cache or local memory. This feature of parallel machines requires programs to access local memory as frequently as possible for efficient computation. Presence of multiple levels of memory forms a memory hierarchy in some machines. HPF considers only one level of memory hierarchy explicitly.

3 High Performance Fortran

HPF consists of extensions to Fortran 90. One advantage of building a new language on Fortran is that it is expected to facilitate reuse of the enormous inventory of existing codes written in Fortran 77 (or earlier versions). In addition, this approach leverages existing skills of experienced Fortran programmers, as well as the significant investment of computer manufacturers, which have developed highly-optimized Fortran compilers. This section includes a brief overview of Fortran 90 and the major extensions to Fortran 90 contained in the HPF specification. Complete details concerning HPF may be found in references [18, 25].

3.1 Fortran 90 Features

Although Fortran 90 includes major extensions to Fortran 77 in a broad range of areas, the principal features of interest in the present paper involve array processing. In this area, notable extensions include:

- Processing of arrays as individual data objects. For example, the Fortran 90 statement '`A = B + C`' adds corresponding elements of arrays '`B`' and '`C`' without requiring do-loop indexing over each element of the arrays.

- New syntax for array sections, which contain a subset of elements from an array.
- Masked array assignments, which permit selective processing of array elements. For example, the Fortran 90 statement '`WHERE(A=0) B=C`' assigns elements of the array '`B`' to be equal to corresponding elements of the array '`C`' only for those locations in which the corresponding elements of the array '`A`' are zero.
- Use of arrays and array sections as arguments to a broad case of elemental intrinsic functions *e.g.*, '`B=ABS(A)`' sets each element of array '`B`' to be equal to the absolute value of the corresponding element of array '`A`'.
- New intrinsic functions that perform transformations or reductions on arrays or array sections. Examples include '`SUM()`', which returns a sum of array elements and '`MAXVAL()`', which returns the value of the largest array element.

The processing and manipulation of arrays as individual data objects allows the programmer to represent bulk data parallelism in a form that can be exploited relatively easily.

3.2 The HPF Extensions to Fortran 90

The HPF extensions provide the compiler with information about locality as well as concurrency present in the application code. This information is of utmost importance in achieving high performance on distributed memory machines. These extensions fall into four categories: compiler directives, new parallel constructs, library routines, and an escape mechanism for interfacing with other languages and libraries[1].

3.2.1 Compiler Directives

Compiler directives are structured comments that suggest implementation strategies or assert facts about a program to the compiler. They may affect the efficiency of the computation performed, but they do not change the semantics of the program. Analogous to Fortran 90 statements, there are declarative directives, to be placed in the declaration part of a scoping unit, and executable directives, to be placed among the executable Fortran 90 statements. Efficient execution of data-parallel applications on distributed memory machines requires distribution of data such that data locality and/or load balance is achieved. HPF data alignment and distribution directives allow the programmer to advise the compiler how to assign data objects (typically array elements) to the memories of processors.

HPF uses a two-level mapping of data objects to 'abstract processors'. Objects (typically arrays) are first aligned relative to one another, and then groups of aligned objects are distributed onto a programmer-defined rectilinear arrangement of abstract processors (using the

[1]HPF also restricts sequence and storage association available in Fortran 90 due to incompatibility with the data distribution directives (see [18] for further details).

'PROCESSORS' directive). The final mapping of abstract to physical processors is not specified by HPF and is implementation dependent.

The 'ALIGN' directive can be used to enforce the co-locality of different arrays. All array elements aligned with an element of the template are mapped to the same processor. The template can either be provided explicitly by using the 'TEMPLATE' directive or implicitly by using another array as a template.

The template of an array is distributed among the set of abstract processors by using the 'DISTRIBUTE' directive. Several patterns of partitioning are available. The important ones are 'BLOCK' and 'CYCLIC'. The former slices the template elements into uniform contiguous blocks of elements, while the latter assigns the template in a round-robin fashion. In addition, a particular dimension of the template may be collapsed or replicated onto the abstract processor grid.

Figure 1 gives an example of the use of the above HPF directives for distributing an 8×8 matrix amongst four processors.

The combination of alignment information of an array and distribution information of its template defines the mapping of an array. However, this mapping need not be fixed throughout the code's execution. Instead, distributed arrays may be remapped at any place in the code —although this obviously results in large data movements and should generally be avoided. Distributed arrays which are remapped during runtime are first flagged to the compiler with the 'DYNAMIC' attribute. The actual remapping is then triggered by executing the directives 'REALIGN' or 'REDISTRIBUTE', as illustrated in the code fragments in §6. Note that in the new HPF-2.0 specifications (see §3.5) data mapping with these directives have now been moved from the core language definition over to the list of *approved extensions*, in recognition of the efficiency problems their use causes. However, the 'TRANSPOSE()' function still allows arrays to be remapped, albeit in a very limited manner.

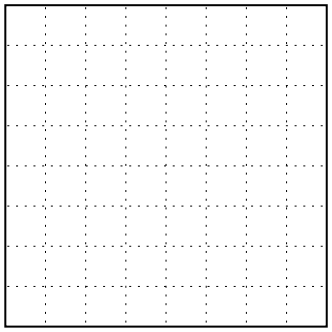
3.2.2 Parallel Constructs

Parallel constructs are provided in HPF to enable the programmer to make explicit assertions about concurrency in a code. Available features include the 'FORALL' statement, the 'FORALL' construct and the 'INDEPENDENT' do-loop directive.

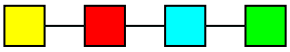
The 'FORALL' statement can be used to express assignments to sections of arrays. It is similar to the array assignment of Fortran 90, but allows more general sections and computations to be specified. The semantics of 'FORALL' guarantee that array elements may be assigned in an arbitrary order. To ensure determinism it is required that each array element on the left hand side (LHS) is assigned only once. The execution of the 'FORALL' statement may require intra-statement synchronization; the evaluation of the left side expression of the 'FORALL' assignment must be completed for all array elements before the actual assignment is made. A sequence of 'FORALL' statements can be combined using the 'FORALL' construct with similar semantics.

The 'INDEPENDENT' directive asserts that the statements in a particular section of code do

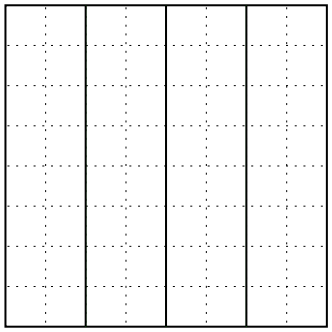
```
(a)       REAL,DIMENSION(8,8) :: ar
```



```
(b) !HPF$ PROCESSORS :: procs(4)
```



```
(c) !HPF$ TEMPLATE,DIMENSION(8,8) :: tplate
    !HPF$ DISTRIBUTE (*,BLOCK) ONTO procs :: tplate
```



```
(d) !HPF$ ALIGN WITH tplate :: ar
```

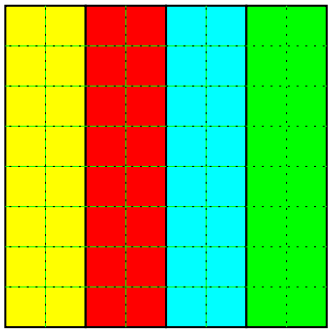


Figure 1: Example of the use of HPF directives in distributing an array. (a) The array to be distributed: '`ar(8,8)`'; (b) The directive for specifying the abstract processor layout, here a linear chain of four nodes; (c) The directive for specifying the (zero-storage) '`TEMPLATE`' array '`tplate(8,8)`' which is then distributed with the first index dimension local within a processor, and the second index dimension '`BLOCK`' spread across the processors; and (d) By aligning the data array '`ar()`' with the template object '`tplate()`', the data array is distributed across the four processors.

not exhibit any sequentializing dependencies. Although it does not change the semantics of the code, it can be used to provide information to the language processor in order to allow optimizations. This directive is useful in cases when traditional dependency-analysis methods may be extremely difficult, especially because of indirections.

3.2.3 Intrinsics

Experience with massively-parallel machines has identified several basic operations that are very valuable in parallel algorithm design. HPF has added several classes of parallel operations to the operations already provided in Fortran 90. Important HPF computational intrinsics (including the Fortran 90 intrinsics) are the following:

1. *Simple reduction functions*: These functions can be used to calculate summations and other related operations along one or more dimensions of an array.
2. *Combining scatter functions*: These allow for combining non-overlapping subsets of array elements.
3. *Prefix and suffix functions*: These functions provide scan operations on arrays and subarrays.
4. *Array sorting functions*: These functions can be used for sorting in increasing or decreasing order.
5. *Array manipulation functions*: Many useful operations such as transposing an array or rotating an array are available.
6. *Array location functions*: These are useful in finding location of elements in arrays with maximum or minimum value.
7. *Array construction functions*: These allow construction of new arrays, potentially of new size and shape, by using the elements of another array.
8. *Vector and matrix multiplication functions*: These include dot-product and matrix multiplication.
9. *Elemental intrinsics functions*: These functions act on individual elements of an array.

In addition, several system inquiry functions useful for controlling parallel execution are provided.

3.2.4 Extrinsics

An escape mechanism is provided in the form of an interface which can be used by other languages and possibly supporting other programming paradigms, including explicit message-passing. The HPF library also provides a set of inquiry functions that return all necessary information to fully exploit the actual data distribution generated by the HPF program.

3.3 Anatomy of a typical HPF compiler

In this section, we briefly discuss the important issues for a typical HPF compiler. This section is limited to parallelization of explicit parallelism available for which we have had extensive experience [4] and is the main focus of this paper. It is a very brief and simplistic overview of an HPF compiler.

Efficient parallelization of '`FORALL`' statement, '`FORALL`' construct and array assignment functions on distributed memory machines can be decomposed into three parts

- Computation Partitioning
- Recognition/Generation of Communication
- Performing the Communication generated

We will assume that computation for a given iteration of loop or array assignment would be performed based on the owner computes rule (based on the LHS) unless otherwise specified. For the second step, the compilers needs to derive the communication (if any) required for the parallelization of the parallel constructs. This requires information about the distribution of variables on the Left Hand Side, Right Hand Side and loop index bounds. A typical HPF compiler would generate a send processors list, receive processors list, local index list of the source array(s), and local index list of the destination array. There are two ways of determining the above information. 1) Using a preprocessing loop to compute the above values or, 2) based on the type of communication the above information may be implicitly available, and therefore, not require preprocessing.

The communication generated by most of the constructs is collective in nature. A compiler can generate send/receives (or other low level constructs) for performing the communication. However, there are three important reasons for using collective communication:

1. *Improved performance and scalability*: To achieve good performance, interprocessor communication must be minimized. By developing a separate library of interprocessor communication routines, each routine can be optimized.

2. *Increased portability:* Separating the communication library from the basic compiler design enhances portability because only the machine specific low-level communication calls in the library need to be changed. Note that the compiler communication library can be optimized for each machine and portability is derived from a common interface so that the code generation by the compiler does not have to change.

3. *Improved performance estimation.* Future compilers would require a capability to perform automatic data distribution and alignment. Distribution of temporary arrays must be determined by the compiler. Such techniques usually require computing trade-offs between exploitable parallelism and the communication costs. The costs of collective

communication routines can be determined more precisely than generating point to point communication for each pair of communicating processors, thereby enabling the compiler to generate better distributions.

A list of important communication primitives required are as follows [4].

- **Transfer:** Single source sends data to a single destination.
- **Broadcast**: One processor (source node) sends data (scalar variable or multi-dimension array) to all processors or a subset of processors (destination nodes).
- **Reduction**: It is the inverse of a broadcast. Typically a combining function is used to combine the inputs from all the processors.
- **Overlap_shift:** Shifting data into overlap areas in one or more grid dimensions. This is particularly useful when the shift amount is known at compile time. This primitive uses that fact to avoid intra-processor copying of data and directly stores data in the overlap.
- **Concatenation:** This primitive concatenates a distributed array and the resultant array ends up in all the processors participating in this primitive.
- **Many-to-many Communication:** Each processors sends a different size message to a subset of all the processors.

Several algorithms have been studied in the literature for the above communication structures for different interconnection networks and are part of standard textbooks on parallel computing.

Some of the features which the communication operations have to support are:

- Many of the operations have to be performed only along a few dimensions; *e.g.*, a multicast requires sending a message to the processors along one dimension of a two dimensional mesh. This requires a compact representation of participating processor lists in each of the broadcasts.
- There is a need for moving array sections. Thus, the message in general is as an array section (with block-sized, stride along each dimension). The communication primitives available for earlier versions of the communication libraries of parallel machines support only contiguous blocks of data. When array sections have to be communicated, routines are required to pack and unpack data into and from a contiguous buffer. These routines pack data into a contiguous block before passing it to communication network, and unpack data and place it to the right location upon its arrival of destination processor.

The Message Passing Interface (MPI) standard [34] addresses many of the above issues.

Parallelization of the independent statements may require careful mapping of the iterations to minimize computations as well as load balance. In this paper, we will limit the use of

'INDEPENDENT' do loops to cases in which each iteration can be assigned to arbitrary processors such that approximately equal number of iterations is performed on every processor or assigned using the owner computes rule.

Parallelization of computational intrinsics require tailored libraries to achieve high performance. An HPF compiler will detect calls to intrinsic functions in the HPF program and replaces them with calls to these routines. When an array is passed as an argument to an intrinsic function, it is also necessary to provide some other information such as its size, distribution among the nodes of the distributed memory machine etc. All this information can be stored in a table called a Distributed Array Descriptor (DAD) and passed as another argument to the intrinsic function. The information which is part of this descriptor includes the distribution type of the array, the size of the local array, the size of the overlap areas (if any) etc.

3.4 HPF and Fortran 95

Fortran has, of course, been evolving since its birth in 1954. Hence shortly after the release of the Fortran 90 specifications, work began on the Fortran 95 standard to refine and clarify the work of the previous standard. The preliminary draft of this new standard is currently being debated, but one can identify the following changes which will have an impact on the HPF language.

First of all, the HPF 'PURE' attribute for functions and subroutines have been adopted. This guarantees such procedures to have no adverse effect on data outside those explicitly defined in the interface blocks, and as such can be used to operate on distributed data in parallel. In addition, the HPF 'FORALL' statement and construct has been adopted, enlarging the current computations available with array expressions and the 'WHERE' statement/construct. Fortran 95 also has generalized the syntax for 'WHERE' and 'FORALL' so that they may now be nested together, and expanded the syntax for the 'WHERE' construct to allow masks for the 'ELSE WHERE' clause.

On the otherhand, it currently seems that the HPF procedure attribute 'INTRINSIC()' (for selecting subprograms written in a different language such as serial Fortran 90), and many classes of HPF intrinsic functions (such as for scatter-gather operations) will not be elevated into the Fortran 95 standard.

3.5 HPF-2.0 —The Next High Performance Fortran Standard

The round of meetings for the next release of the HPF standard concluded shortly before the conference *Supercomputing-96* in Pittsburgh. This subsection discusses the improvements made, based upon the document (version 2.0.β) released in mid-November, 1996 [20]

The round of meetings for HPF-2.0 was motivated by three needs: to make corrections and clarifications to the original definition, to provide any new features requested by the HPF community, and to recognise that compiler vendors have to provide efficient implementations of

the full language specification. In response to these issues, the language has been organised into two parts: the HPF language itself, and a set of approved extensions. The latter are described in separate chapters in the HPF standards document, and actually contain some features that were part of the HPF 1.1 standard (for example, the 'REDISTRIBUTE' directive). The rationale being that whilst HPF-2.0 is a complete language, it has also been streamlined to contain features which can be efficiently implemented within a year (as estimated with the vendors' agreement) of the release of the language; whilst the list of approved extensions refer to features which have been debated and approved by the forum, but are acknowledged to be harder to implement efficiently. Also, note that the definition for subset HPF has been left invariant from the HPF-1.1 standard, and moreover that it is still a subset of the HPF-2.0 standard.

HPF-2.0 language features that were not in the HPF-1.1 standard are:

- Data Parallel Execution: The clause 'REDUCTION' has been added in the 'INDEPENDENT' do-loop directive for declaring that a given variable will be updated during that loop. Currently, only a certain set of reduction operations and functions are supported, not arbitrary user-defined ones.

- New intrinsic functions in the 'HPF_LIBRARY' module: 'SORT_DOWN', 'SORT_UP' for sorting the elements in an arbitrary-ranked array.

As well introducing new features, the list of approved HPF-2.0 extensions also contain constructs from the HPF-1.1. The approved extensions are listed in the following subsections.

3.5.1 Data mapping

- The HPF-1.1 directives 'REDISTRIBUTE', 'DYNAMIC', and 'REALIGN', distribution of pointers (*i.e.*, arrays with declared with the attribute 'POINTER'), have been moved here.

- Objects may be distributed onto a subset of the processors. This would be invaluable for multi-resolution codes, such as in the multigrid algorithm. Processor subsets are declared with the Fortran 90 triplet array notation.

- Arrays of Fortran 90 derived datatypes may be distributed, or the elements within an array of Fortran 90 derived datatypes may be distributed, but not both. Hence it is illegal to have a distributed array containing elements which are in turn distributed.

- New distribution formats: 'GEN_BLOCK' and 'INDIRECT'. 'GEN_BLOCK(S)' allows takes an array parameter 'S' describing the *number* of elements in a given array which should be distributed to a given processor. 'INDIRECT(M)' uses an array argument which specifies which processor number (from '1' to the value returned by the intrinsic enquiry function 'NUMBER_OF_PROCESSORS()') a given distributed array element should reside on. Hence the extent of the mapping array 'M' must match that of the array to be distributed.

- The new subprogram interface directive 'RANGE', for use in transcriptive and 'INHERIT' mappings, and informs the compiler to expect one of a possible set of array distributions.
- A 'SHADOW' keyword added to some of the distribution formats, for telling the compiler the 'depth' of any stencil operations in the code. This allows a compiler to set the optimal shadow width in domain decomposition codes.

3.5.2 Data and Task Parallelism

- The 'ON HOME()' directive. This provides advice to the compiler on the data locality of a block of code by informing it which processor should compute which part of a computation, based on the distribution of an array argument. Note that specifying processor subsets are allowed.
- The 'RESIDENT()' directive. This is used as an additional clause to the 'ON HOME()' directive, and is an assertion that the stated list of variables will remain local throughout the computation and will not require communications.
- The 'TASK_REGION'...'END TASK_REGION' directives. These bracket a region of code containing a number of 'ON HOME()' code blocks. This then provides information to the compiler that these 'ON HOME()' sub-regions may operate over disjoint processor subsets, and hence maybe able to execute in parallel.

3.5.3 Extensions to the HPF Intrinsic and Library Procedures

Many of the mapping enquiry functions have been upgraded to reflect the additional mappings allowed. In addition, the intrinsic function 'TRANSPOSE()' can now transpose an array of arbitrary rank.

3.5.4 Miscellaneous

Other Extensions approved in HPF-2.0 are:

- Asynchronous I/O represents a departure from the single-threaded computation model, but is a welcomed acknowledgement of the imbalance between I/O and CPU speeds in modern machines. This feature will allow a separate thread to be created for each I/O operation that has been flagged with a new control specifier ('ID='), that is then joined back onto the main control thread with the 'WAIT()' statement: this pauses the execution until the I/O transfer has been completed.
- Generalisation of the HPF-1.1 extrinsic libraries 'HPF_LOCAL' and 'HPF_SERIAL' for different computational models and languages. For example, there is now the extrinsic library 'FORTRAN_LOCAL' to support interactions between Fortran 77 and HPF subprograms.

4 Application Classification

It is convenient to use the classification of problems into embarrassingly parallel, synchronous, loosely synchronous, asynchronous, and metaproblems [11, 13].

1. The class of embarrassingly parallel problems require little communication or synchronization and can be parallelized using straightforward approaches.

2. In synchronous problems, we have data parallelism with: identical algorithms applied to each point in the data domain, plus identical data dependency patterns between the data domain points.

3. Loosely synchronous problems are data parallel but different data points evolve differently. Points are also often connected in an irregular, data-dependent manner.

4. Asynchronous problems are very irregular and require tailored parallelizations.

5. The class of metaproblems have two or more subtasks; each of the subtasks belong to one of the above categories. Each of the subtasks in such an application can be parallelized individually, and there may be parallelism between them.

In the following sections, we provide an analysis of important applications which can be used for solving applications using HPF. Most of the codes presented are kernels from real applications and are provided to show the specific HPF construct or directive. However, we should stress that due to the range of codes at our disposal, and the restriction of the HPF 1.1 standard implemented by today's compilers, the applications here are largely embarrassingly parallel or synchronous in nature. The need for more general task-parallelism constructs and irregular mapping of data has been addressed in the HPF 2.0 standard [20], following feedback from the initial HPF-1.0 process [19]. These shortcomings in HPF-1.1 have impacted work in General Relativity simulations [24], where the problem mesh is irregular, and where the simulation's adaptive mesh refinement algorithm causes load imbalances.

5 Embarrassingly Parallel Applications

This class of applications require execution of several independent instances of the problems and can be parallelized relatively easily. This information about independence can be provided to the compiler by using the '`INDEPENDENT`' directive. At the end of the computation of statistical properties of several runs needs to be evaluated. This can be done by use of the reduction, gather/scatter intrinsic functions.

5.1 Conversion between Random Numbers

Many applications require the use of random numbers with nonuniform distribution. The system provided random numbers are typically generated using a uniform random number generator and a simple algorithm like the Box-Muller method can then be used to generate a set of Gaussian distributed deviates this set. This code assumes that the system random number generator (or some other) can supply a vector of uniform deviates, and applies a vectorized Box-Muller algorithm to obtain Gaussian deviates. Such a generator is readily available from the 'RANDOM_NUMBER' Fortran 90 intrinsic function.

This algorithm uses an accept/reject approach and not all generated random number pairs are useful. Statistical fluctuations in the accept/reject mechanism require extra deviates to be computed. A simple approach is to generate a larger (say twice) the number of potential deviates as are requested by the caller. Since, the Box-Muller algorithm has an intrinsic efficiency of about 2.0/1.27, this would typically be large enough for a large set.

The NAS EP ('Embarrassingly Parallel') benchmark [2] in appendix A.1 follows the Box-Muller approach, converting vectors of uniform-deviate random numbers into vectors with a Gaussian distribution. However, because the benchmark specifies how these numbers are generated, the system-dependent random number generator has to be replaced with a portable implementation. In addition, the EP benchmark is typified by the need to histogram or tabulate random deviates with a particular correlation property. The code illustrates how this might be achieved compactly in HPF. A distribution template is set up to spread samples from some random process across processors. Temporary arrays are set up to allow processors to make a partial summations and counts of those samples which fail some test in their own local memory.

The 'WHERE' construct is used to express the potential independent processor operation to partially sum the samples. The 'SUM' intrinsic function can then be used to sum the partial summations.

However, there is a potential problem with the above approach because packing generates unnecessary communication. In a message passing implementation each processor would generate its own set of random numbers without any communication. This problem can be solved by using a code that generates P independent streams of random numbers (where P is the number of processors). Each stream generates enough random numbers such that enough Gaussian deviates are generated. This can be done using a do loop with the 'INDEPENDENT' directive.

5.2 Data Analysis

Several applications require the use of statistical analysis over several measurements. High energy physics applications require such analyses over a very large number of events. These applications can make use of the HPF features:

1. 'INDEPENDENT'

2. Reducing intrinsics such as 'SUM', and 'SCATTER/GATHER' for histogramming.

In addition, the HPF-2.0 [20] specifications have added the 'REDUCTION' clause has been added to the 'INDEPENDENT' directive to allow more generic reduction operations to be written (see §3.5). In particular, a given variable is marked down by this clause as to be only accessible in a reduction-like operation statement (such as 'X = X + A(I)'). However, the range of reduction operations allowed are currently limited to:

- Arithmetic binary operators: '+', '-', '*', '/'
- Logical operators: '.AND.', '.OR.', '.EQ.' and '.NE.'
- Reduction intrinsic functions: 'MAX', 'MIN', 'IAND', 'IOR', 'IEOR'.

We demonstrate the use of HPF for such statistical analysis using the calculation of Hough transform. The general technique of Hough transform is computationally very intensive and has been used to detect lines, circles and arbitrary shapes in images. In the following we briefly describe this technique for detecting lines.

The general equation of a straight line can be given by the parametric equation:

$$r = x\cos\theta + y\sin\theta \tag{1}$$

where θ is the angle that the normal to the line given by equation 2 makes with the x axis, and r is the length of the normal. Any edge point (x_i, y_i) on this line satisfies the equation:

$$r = x_i\cos\theta + y_i\sin\theta \tag{2}$$

The above equation represents a sinusoidal curve in the (r, θ) space. Any point on this curve corresponds to a line passing through (x_i, y_i). Thus, the curves corresponding to all the points on a line in the (x, y) space must intersect at the same point in the (r, θ) space. Each edge point contributes '1' to the (r, θ) cells given by equation 2 and the cells corresponding to local maxima give the desired lines.

Thus, each edge pixel votes for bins in (ρ, θ) space. A bin which receives a large number of votes represents the presence of a line (or other feature) in the image. The code in appendix A.2 illustrates how this might be achieved compactly in HPF. The input to this program is and edge image (a pixel is '1' if it is and edge pixel). This is typically derived by using simple template matching algorithms [3]. The first phase of the code packs all the edge pixels in a compact array. This removes a conditional check which will otherwise be required to determine whether this pixel can vote for the resultant Hough array. Edge pixels are non-uniformly distributed in the edge array. This also leads to a load balancing for the voting stage.

The next stage of the algorithm creates a voting array of size equal to the number of pixel times the number of angles. This array is used to determine the resultant Hough array by using a 'COUNT_SCATTER()' intrinsic.

Also, for realistic images suitable runtime support and redistribution of edge pixels can be used to reduce the communication overhead generated by the 'COUNT_SCATTER()' [28]. The 'COUNT_SCATTER()' may generate hot spots if most of the votes are for a few bins in the Hough array.

5.3 Unindexed Search

Several application require searching of a specific pattern in a database. The type of search depends on whether the data is stored using indexes or unindexed. In the former case, the index could be used to accelerate the search process. The latter requires searching the whole database. Typically an unindexed search of full text held in the form of many documents. Each entry in the document database is abstracted with a sophisticated algorithm. These compressed files are then compared to the query (such as "Find me all newspaper articles on Mickey Mantle") in parallel and a match score extracted. The scores are sorted (or thresholded) and the user receives a list in order of decreasing (or thresholded) scores.

These applications would require use of a do loop with an 'INDEPENDENT' directive or a 'FORALL' loop for comparing the pattern with all the documents. This would be followed by a sorting operation ('GRADE' function in HPF) to order the documents based on the quality of the match. Additional features may have to be added to HPF for execution on larger data sets to improve the efficiency of I/O required.

5.4 Methods for Stock Pricing Models

Stock options are contracts that give the holder of the contract the right to buy or sell the underlying stock at some time in the future. Option contracts are traded just as stocks are traded, and models that quickly and accurately price option contracts are valuable to traders and financial managers. Speculators participate in the option market to capture potential high profits with relatively small investment capital. Financial managers buy and sell options to hedge risk in their investment portfolios.

A binomial model uses binary trees to represent possible up/down movements in asset price over the life of an option contract, and are more efficient than Monte Carlo methods. Finucane ([9]) demonstrated that binomial methods provide comparable price estimates as Monte Carlo models for market observations with known model parameters. The computational structure of these models can be represented as a binomial tree. There are three phases in the American stochastic volatility model:

1. The tree starts with a root. At each time step, a leaf node of the tree generates two nodes. Thus, at each time step, the number of levels of the tree increases by one. The computational structure guarantees that the value of each child depends only on its parent. This is repeated for several time steps.

2. The call values at each of the leaf nodes is calculated.

3. A reverse binomial tree is executed to collect the values at the root node.

Only the values of leaf nodes are useful in later computations. The second step is embarrassingly parallel. The last step is the inverse of the first step. The first step can be parallelized as follows:

1. Declare a two dimensional array of size $p \times (2^{levels}/p)$.

2. The tree is grown for $\log(p)$ iterations using '`FORALL`' with appropriate masks for operating on the first row of the matrix. This would in general result in communication.

3. Each column stores a subtree of the binomial tree for the remaining iterations. Using a block distribution would ensure that no communication is generated.

The program fragment which uses the above strategy is given in appendix A.3.

Another approach relies on the direct stochastic integration of the Langevin equation for the security price process [31]. It generates the probability distribution of security prices using a Metropolis algorithm. Similar Monte Carlo methods have been successfully used in many financial modeling applications and are notoriously computationally intensive. The basic computational structure requires calculating the probability of transition from one path to another. Transition probability is a product of local transition probabilities of the segments of a path, which are in turn related to the local curvatures of the paths.

The algorithm has several levels of parallelism each of which can be exploited in HPF:

1. The same code is run for all stocks independently.

2. For each stock there are several parameters of interest (the same ones for each stock) which can be computed independently.

3. For each parameter a path integral has to be obtained of length proportional to the number of time steps. All the odd (and even) time steps can be updated concurrently

This code can be easily represented in HPF using '`INDEPENDENT`' loops and '`FORALL`' statements.

5.5 Summary

A summary of important HPF features required for the different embarrassingly parallel applications is provided in Table 1. From the point of view of a user with existing Fortran 77 code, casting the problem as embarrassingly parallel is especially attractive as this would be the simplest means of exploiting parallelism in the algorithm. In the simplest case, this might

Application	F90/HPF Features
Binomial Stock Pricing (A.3)	Array syntax, 'WHERE', 'EOSHIFT()'
Box-Muller Gaussian random numbers (A.1)	Array syntax, 'WHERE', 'SUM()'
Hough image transformation (A.2)	Array syntax, 'FORALL', 'PACK', 'COUNT_SCATTER()'

Table 1: Embarrassingly parallel applications discussed in this paper with the appendices where their HPF implementations can be found, and the key F90/HPF features which they exploit.

just be declaring the computational heart of the code as a HPF 'PURE' subroutine, which is then called from within an 'INDEPENDENT' declared do-loop.

It should also be noted that there are provisions within HPF for applications to be written in a more traditional SPMD with message passing calls paradigm, via the use of the 'HPF_LOCAL' extrinsic library call. Inside this call, non-distributed data are no longer shared across all the processors, but instead each processor holds its own —in general variant— version of that data.

6 Synchronous Applications

HPF is targeted largely towards the solution of synchronous problems where the decomposed dataset is regular (has identical data dependencies), data locality is maintained, and the same operation is performed on all data points. These problems use multidimensional arrays as the basic data structures which can be manipulated by the explicit parallel constructs as well as by the intrinsics described in the earlier sections. Data-locality and/or load balancing information can be provided by distribution directives such as 'BLOCK' and 'CYCLIC'. For most applications, a program can be decomposed into several stages. It is important to observe that real industrial applications codes involve more than one basic algorithm to be applied to the problem data. Depending on the data access patterns, the layout of data required for one phase may be different than the other [16]. For this reason the data remapping features of HPF are particularly important —although these are not in the subset HPF standard [18] nor in the core HPF-2.0 specifications (see §3.5). In HPF this is done directly with the 'REDISTRIBUTE' and the 'REALIGN' mapping directives; upon the arrays being re-mapped first begin given the HPF 'DYNAMIC' attribute. So, for example, the following code fragment will redistribute an array from a 'BLOCK' to a 'CYCLIC' distribution:

```
      REAL,DIMENSION(N) :: array
!HPF$ DYNAMIC :: array
!HPF$ DISTRIBUTE (BLOCK) :: array
      ... Computation with BLOCK distribution on array ...
!HPF$ REDISTRIBUTE (CYCLIC) :: array
      ... Computation with CYCLIC distribution on array ...
```

alternatively, it may be more convenient to do the redistribution by realigning the designated array with another object, viz:

```
      REAL,DIMENSION(N) :: array
      REAL,DIMENSION(N) :: dist1, dist2
!HPF$ DISTRIBUTE (BLOCK) :: dist1
!HPF$ DISTRIBUTE (CYCLIC) :: dist2
!HPF$ DYNAMIC :: array
!HPF$ ALIGN WITH dist1 :: array
      ... Computation with BLOCK distribution on array ...
!HPF$ REALIGN WITH dist2 :: array
      ... Computation with CYCLIC distribution on array ...
```

An alternative to explicitly redistributing the above arrays is to use the Fortran 90 '`TRANSPOSE()`' intrinsic function, as is illustrated in the Fast Fourier transformation case in § 6.8.4 and the ADI algorithm in §6.6.1. This could be useful in algorithms where computations act in alternating directions of the, as for example in the x and y directions of a two dimensional image. Unfortunately this is only defined for rank-2 matrices, but a generalization for other larger ranks have been proposed for the HPF 2.0 standard [20].

6.1 Full Matrix Algorithms

Algorithms involving matrices and vectors are applied in several scientific and non-scientific applications. Matrices can generally be classified into dense or sparse matrices. The former contain few or no zero entries. The latter class deals with matrices where a majority of elements are zero. This sparsity can be exploited to achieve better algorithms such as the conjugate gradient method; the suitability of the current HPF-1.1 standard for which have been examined [7]. The shortcomings for sparse matrix problems are largely to do with load distribution for solving matrices with arbitrarily sparse structures (see the discussion on regular and irregular problems in §1), and have better addressed with the approved extensions in HPF-2.0 (see §3.5).

Examples of full matrix algorithms include Gaussian elimination, Cholesky factorization and matrix multiplication. The parallelism available in many of the full matrix algorithms is structured and HPF can be effectively used to represent the concurrency available. Further, effective parallelization of these applications on coarse grained machines require block, cyclic or block-cyclic distributions along one or more dimensions. Matrix Multiplication and Vector Dot Product are also part of the HPF intrinsic library.

For example, the following is a code fragment from Gaussian elimination, and can be effectively parallelized using a cyclic distribution.

```
      do k = 1, N
          . . .
          forall(i=1:N, j=k:NN, indx(i).EQ.-1)
     &          a(i,j) = a(i,j) - fac(i) * row(j)
      end do
```

However, getting high performance for these applications requires careful fine tuning for a particular architecture. For example, efficient implementation of Gaussian Elimination may involve pipelining. This may depend on the underlying communication network. Effectiveness

of hardware tailored libraries have been clearly demonstrated for these algorithms where order of magnitude improvements can be obtained over simplistic code. Although these applications can be effectively used to demonstrate important features of HPF and modest performance can be achieved, it is our belief that many of them would be available as libraries. HPF, with a rich set of data distribution directives and interfacing functions, should be able to provide an easy parallel interface to these libraries.

6.1.1 Gaussian Elimination

The Gaussian elimination algorithm lies at the heart of many direct solvers for general non-symmetric matrices. Appendix A.4 gives a code which performs this operation, for an example matrix with element generated by the Fortran 90 random number intrinsic routine. The aim is to factorise a matrix A into upper and lower triangular parts, from which solutions to the matrix equation $Ax = b$ can be solved for arbitrary right hand side vectors b.

The parallelism in the code comes through distributing columns of the problem array across the processors. This is done in a cyclic manner since as the algorithm progresses, fewer number of columns become involved with the computation, and hence the balance of load will be maintained longer than with the block distribution (as illustrated in figure 1).

Stability of the algorithm is ensured by the use of partial pivoting is performed by the code, which requires a search for the location of the largest absolute value of an element along a given matrix column. This is done by the Fortran 90 intrinsic '`MAXLOC()`' with a '`MASK`' argument to ignore the rows which have already been computed upon. Once the pivot has been found, the computation involves an update of the a part of the matrix with the outer product of the pivot column and pivot row vectors —this computationally intensive part is where much of the available parallelism is exploited.

6.1.2 Matrix and Vector Multiplications

Fortran 90 provides two intrinsic functions for the multiplication of matrices and/or vectors: the scalar function '`DOT_PRODUCT()`' for evaluating the inner product of two vectors, and the array function '`MATMUL()`' for more general arrays. In theory, the compiler vendors should be able to implement these intrinsics more efficiently than hand-coding by the users. This subsection will discuss how these are used via a few of the more common BLAS routines, and how the arrays should be distributed in the context of HPF.

BLAS 1 Multiplications

This class of operations are between vectors, and produce a single scalar for the result, as typified by the functions '`xDOT`'. In terms of HPF, it is obviously best for the two vectors to be distributed equally amongst the processors, and aligned with respect to each other. The sum over the multiplied elements should then be the only part requiring communications. In principle, the function '`MATMUL()`' can be used, but the function '`DOT_PRODUCT()`' would remove

the need to copy a 1×1 array to a scalar.

BLAS 2 Multiplications

These operations are between a matrix and a vector, or the outer product of two vectors. In either cases, this is done with the intrinsic function '`MATMUL()`'.

For BLAS operations of the type '`xGER`' where a matrix A is given by the outer product of two vectors xy^T, the most computationally efficient distribution would be for A to be distributed, but x and y to be replicated over all the processors. Hence the code fragment may look like:

```
      REAL,DIMENSION(N) :: x,y
      REAL,DIMENSION(N,N) :: A
!HPF$ DISTRIBUTE(*,BLOCK) :: A
      A = MATMUL( RESHAPE(x,(/N,1/)), RESHAPE(y,(/1,N/)) )
```

which should require no communications. This would obviously not hold if for memory reasons either one of the two vectors has to be distributed. Note that $\vec{x}$ and $\vec{y}$ could not be used directly by the '`MATMUL()`' intrinisc; instead, they have to be '`RESHAPE()`'-ed so as to become $N \times 1$ and $1 \times N$ matrices, respectively. In this way, the function can be told how the two vectors should be combined.

For BLAS 2 operations like '`xGEMV`' where a matrix A is multiplied with a vector x to produce a vector y, communications can be avoided by distributing & aligning A and y, but with x replicated. With this scheme, each element in y maybe computed in parallel, as in the code fragment:

```
      REAL,DIMENSION(N) :: x,y
      REAL,DIMENSION(N,N) :: A
!HPF$ DISTRIBUTE(BLOCK) :: y
!HPF$ ALIGN (:,*) WITH y(:) :: A
      y = RESHAPE( MATMUL( A, RESHAPE(x,(/N,1/)) ),(/N/) )
```

where the result of the '`MATMUL()`' operation has also been '`RESHAPE()`'-ed so as to translate from an $1 \times N$ row-vector to an N-dimensional vector, to match the shape of $\vec{y}$.

BLAS 3 Multiplications

The class of BLAS 3 operations is typified by the matrix-matrix multiplication routine '`xGEMM`'. One can treat the parallelisation by thinking of this as a generalisation of the matrix-vector case above, such that in the operation $C = AB$, the matrix B is replicated whilst A & C are distributed. This may not be feasible if the matrices are very large, and machine memory limited; in this scenario, one can minimize the communications in the multiplication's inner-loop by distributing B in the other dimension to that which A and C are distributed. This is illustrated in the fragment:

```
      REAL,DIMENSION(N,N) :: A,B,C
!HPF$ DISTRIBUTE(BLOCK,*) :: A,C
!HPF$ DISTRIBUTE(*,BLOCK) :: B
      C = MATMUL( A,B )
```

All three arrays here have the same ranks and shapes, so there is no need to invoke the '`RESHAPE()`' intrinsic.

6.2 Structured Sparse Linear Systems

Most scientific applications represent a physical system by a mathematical model. A continuous domain of the system to be modeled is discretized by imposing a grid or a mesh over the domain. Solution of the model requires obtaining the values of physical quantities at all the points of the domain. Each grid point is typically directly influenced by the grid points within a small neighborhood. The simulation of each single grid point yields a linear equation relating the value of desired variables based on the values of grid points in a neighbourhood. These result in systems of linear equations with matrices which are very sparse in nature.

6.2.1 Single Grid Methods

Stencil methods are widely used in many direct solution methods for partial differential equations as well as in many cellular automaton and other models in computational physics and also for many simple image processing algorithms. The key feature of stencil based applications is their use of relatively localized data on a regular mesh to update individual variables on the mesh.

For example, the very simple Forward Time Centered Space solution of the Laplace Equation involves the discretization of the field variable ψ on a mesh of some dimensionality —we use a 2d mesh here for illustration— and the iterative update of values $\psi_{i,j}$ according to:

$$\psi_{i,j} = \psi_{i+1,j} + \psi_{i-1,j} - 4\psi_{i,j} + \psi_{i,j+1} + \psi_{i,j-1} \tag{3}$$

Equation 3 can be expressed as the code fragment:

```
 psi = CSHIFT(psi,1,1) + CSHIFT(psi,-1,1) +
&      CSHIFT(psi,1,2) + CSHIFT(psi,-1,2) - 4.0 * psi
```

This shows the use of the '`CSHIFT()`' intrinsic function which cyclically shifts the data by ± 1 in the data dimension stated.

In cases where the application calls for non-periodic boundary conditions, the '`FORALL`' statement/construct may be more appropriate. The below fragment shows how such an explicitly parallel construct may be used to set up the update rule for interior points:

```
 FORALL(i=2:m-1,j=2:n-1)
   psi(i,j) = psi(i+1,j) + psi(i-1,j) - 4.0 * psi(i,j) +
&             psi(i,j+1) + psi(i,j-1)
 END FORALL
```

Where additional code sections can be used for the boundary values.

In general the mesh variable data would be distributed across the processors' memories in '`BLOCK`' fashion for such an application. Providing the data array was significantly larger in size than the number of processors, then the actual communications needed to move data across

processor boundaries would be low. Most mesh points would be in fact interior to the processors own memory.

For some applications, however it is necessary to carry out the mesh updates in a particular order. Some examples are the red-black solver algorithm used for diffusion problems or another well know case is that of Monte Carlo models in computational physics where the update algorithm requires updating alternate mesh variables in each dimension at once. Generally this reduces the parallelism by a factor of two, but since mesh sizes are (usually) much larger than the number of processors, this does not affect the *exploitable* parallelism.

HPF provides a suitable mechanism for such a red-black checkerboard update. The '`BLOCK`' data distribution —or '`CRINKLE`' mapping as it was originally known when first implemented on the Distributed Array Processor (DAP) in the early 1980's— allows the data to be 'staggered' across the processors' memories and thus updates to be ordered to suit the applications algorithm while still exploiting data locality.

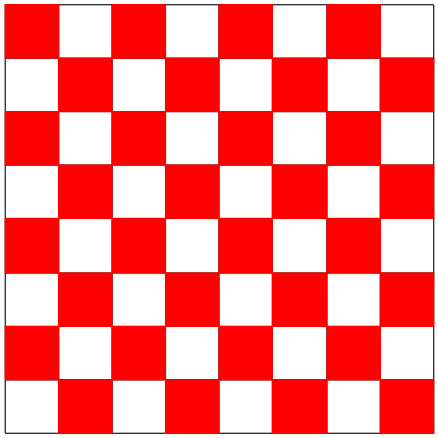


Figure 2: Red-Black mesh coloring.

The following code fragment illustrates how red-black ordering (Figure 2) can be done using a pair of '`FORALL`' constructs:

```
 FORALL(i=2:m-1:2,j=2:n-1:2)
   psi(i,j) = psi(i+1,j) + psi(i-1,j) - 4.0 * psi(i,j) +
&             psi(i,j+1) + psi(i,j-1)
   psi(i+1,j+1) = psi(i+2,j+1) + psi(i,j+1) - 4.0 * psi(i+1,j+1) +
&                 psi(i+1,j+2) + psi(i+1,j)
 END FORALL
 FORALL(i=2:m-1:2,j=2:n-1:2)
   psi(i+1,j) = psi(i+2,j) + psi(i,j) - 4.0 * psi(i+1,j) +
&             psi(i+1,j+1) + psi(i+1,j-1)
   psi(i,j+1) = psi(i+1,j+1) + psi(i-1,j+1) - 4.0 * psi(i,j+1) +
&             psi(i,j+2) + psi(i,j)
 END FORALL
```

where the two separate 'FORALL' constructs are used for the 'red' and 'black' interleaving meshes. Note the use of stride-2 to construct the red-black ordering. In this example only the interior points are dealt with. Separate code would be needed to loop over the edge points to ensure the correct boundary conditions were applied.

A similar stride structure can be used to construct other more complex mesh 'color' groupings, such as are used for Monte Carlo simulations where longer range interactions are involved. All of the above is general for one, two, three or higher dimensional data structures, and the only major changes are the data distribution directives.

6.3 Unstructured Sparse Linear Systems

The applications described in the previous subsection use structured grids to model the underlying physical quantities. The access pattern of each node in the grid is localized and is on a regular mesh. In many applications the access pattern though localized is on a irregular or an unstructured mesh. Example of such applications include iterative solvers using sparse matrix vector multiplications and explicit unstructured mesh fluids calculations.

Consider the solution of a Laplace Equation on an unstructured mesh shown in Figure 3. It will be similar to Equation 3; however, each node of the mesh will potentially have different number of neighbors. Unlike, the structured grids the addressing of these neighbors has to be explicit. This requires the use of two data structures: node array and edge array. It also prevents the direct use of parallel constructs such as the `FORALL statement` and requires the use of a combining `SCATTER` function. In appendix A.5 we describe a code that represents a simple edge sweep on an unstructured mesh [2]. The flux at every node is updated based on the array values corresponding to its neighboring nodes. The neighbor nodes are stores explicitly in a separate array.

The code in appendix A.5 partitions the nodes using a `BLOCK` distribution. Unlike regular grids this may result in a large amount of interprocessor communication. Effective parallelization of these applications require partitioning the nodes such that most of the accesses are local to the processor and the computation on each node is balanced. An example of such a partitioning for four processors is given in Figure 3. The amount of communication generated will depend on the locality of access. The cost of the communication will be low if the nodes are reordered such that most of these accesses are local. This reordering can be potentially achieved in HPF 1.1 by use of extrinsic functions to perform graph partitioning using the edge array [37]. This reordering can also take into account the computational load associated with every node.

[2] A similar kernel code was made available to us by Saltz's group at University of Maryland

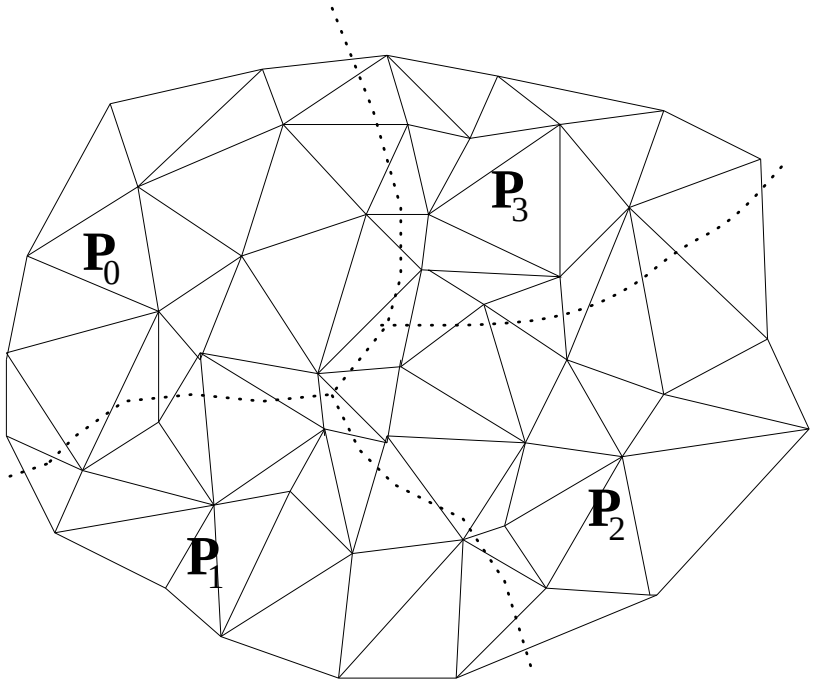


Figure 3: Unstructured Mesh (partitioned for 4 processors)

6.4 Crystalline Monte Carlo

Many simulations in condensed matter physics such as the Ising, Potts and Heisenberg models and calculations in Quantum Chromodynamics use a Monte Carlo statistical mechanical algorithm to explore the phase space. These applications typically involve a regular structure and can be effectively represented using a Fortran multidimensional array, with any parallelism in the update dynamics exploited by distributing the array via HPF directives. The computation would then be expected to make use of the '`FORALL`' and '`WHERE`' constructs/statements, and the '`CSHIFT()`' intrinsic.

6.4.1 Potts Model

The Potts model describes a spin system whereby each site may take on one of Q states, with the local field at a site i computed by the expression:

$$h_i = \sum_{<j>} S_i J_{ij} S_j \tag{4}$$

where the sum '$< j >$' is over the near-neighbour sites. The interaction J_{ij} for this model is non-zero for only when S_i and S_j have the same state; *i.e.,*

$$J_{ij} = J\delta_{S_i,S_j} \tag{5}$$

where $\delta_{a,b}$ is the Kronecker delta function and J is the constant coupling constant.

This algorithm requires ensuring that interacting degrees of freedom are not updated simultaneously. A four-colour variation of the red-black chess board update scheme (cf. §6.2) is used to exploit the near-neighbour nature of the interactions, so as to ensure the update is performed correctly. The HPF/Fortran 90 '`WHERE`' construct is hence used to pick out those sites being considered for updating. The '`MERGE()`' and '`CSHIFT()`' intrinsic functions are used to calculate the local fields according to equations (4) and (5). These concepts are illustrated in appendix A.6 for a simple 2-dimensional Potts model.

A historically important variation of the Potts model is the Ising spin model, where the states of a site take on one of two values, representing a magnetic spin system's 'up' and 'down' state. The Ising model is the exemplar for a whole family for statistical mechanical problems including spin glass and neural network associative memory models. The computational structure of this application is similar to the Potts model.

6.5 Quenching Algorithms

The solution of of 2-D Binary Phase Quenching uses a random binary phase mixture and applies the Cahn Hilliard Cook equation to quench the system into a cooled state. It evolves a random mixed phase configuration, through an intermediate nucleating phase into a quenched two phase system with a quenched slowing down effect.

The Cahn-Hilliard equation can be formulated as follows:

$$\frac{\partial \phi}{\partial t} = m\nabla^2 \left(-b\phi + u\phi^3 - K\nabla^2\phi\right) \tag{6}$$

where the field variable $\phi_{i,j}$ is on a regular mesh as before, but since the equation involves $\nabla^2(\nabla^2)$, the update procedure requires next and next-next nearest neighboring mesh point data.

Some interesting tradeoff situations occur when applications use more complex stencil operations with longer range communications. This can either be formulated explicitly in terms of a single update line using some twenty '`CSHIFT()`' operations as shown in:

```
 phi = phi + dtby2 * (
&      -16.0 * phi
&      + 7.0 * ( cshift(phi, 1,1) + cshift(phi, 1,2)
&              + cshift(phi,-1,1) + cshift(phi,-1,2) )
&            - ( cshift(phi, 2,1) + cshift(phi, 2,2)
&              + cshift(phi,-2,1) + cshift(phi,-2,2) )
&      - 2.0 * ( cshift(cshift(phi, 1,1), 1,2)
&              + cshift(cshift(phi, 1,1),-1,2)
&              + cshift(cshift(phi,-1,1), 1,2)
&              + cshift(cshift(phi,-1,1),-1,2) )
&      - 4.0 * phi ** 3
&            + ( cshift(phi, 1,1)**3 + cshift(phi, 1,2)**3
&              + cshift(phi,-1,1)**3 + cshift(phi,-1,2)**3 )
&      )
```

which is memory-efficient but communications inefficient; or an intermediate data array '`f`' can be used to reduce this to only eight '`CSHIFT()`' operations:

```
 f = 3.0 * phi - ( cshift(phi,1,1) + cshift(phi,-1,1) +
&                  cshift(phi,1,2) + cshift(phi,-1,2) ) + phi ** 3
 phi = phi + dtby2 * ( cshift(f,1,1) + cshift(f,-1,1) +
&                      cshift(f,1,2) + cshift(f,-1,2) -4.0 * f)
```

which sacrifices memory efficiency for better communications performance. This represents a clear tradeoff between memory and communications speed. For a very large size of simulated mesh, memory may be at a premium even in a parallel computer with a lot of distributed memory and the slower communications-intensive algorithm may be preferred. Interestingly, the physics of the problem is such that the natural intermediate variable is a free-energy like quantity and may be of interest as an intermediate calculated quantity in its own right.

The full code for this algorithm can be found in appendix A.7.

6.6 Computational Fluid Dynamics

The solution of Computational Fluid Dynamics (CFD) simulations has traditionally been a major user of high performance computing technologies, from weather predictions [15], flow behaviour around aircrafts [38], to blood flow in a human heart [35]. Hence it is almost inevitable that the HPF language should be explored for its relevance in this arena. The following §6.6.1—§6.6.2 examines the use of HPF for two techniques in the CFD field. Other work has been done in evaluating HPF [12, 39], particularly for where the computational mesh is regular with elements of equal topologies. Moreover, HPF has also recently been used for the large, real-world CFD problem of modelling ocean currents [40], and it is not too unrealistic for the language to replace hand-written message passing codes for parallel CFD applications on regular mesh topologies.

6.6.1 Poisson Equation using ADI

The time dependent Navier-Stokes equation can be written in terms of the vorticity (ζ) and stream (ψ) functions by the coupled equations:

$$\frac{\partial \zeta}{\partial t} = \frac{1}{R}\nabla^2\zeta - \left[\frac{\partial \psi}{\partial y}\,\frac{\partial \zeta}{\partial x} - \frac{\partial \psi}{\partial x}\,\frac{\partial \zeta}{\partial y}\right] \tag{7}$$

$$\nabla^2\psi = \zeta \tag{8}$$

where equation (8) states that we are dealing with the incompressible problem.

A common approach to this problem is to use a *split operator* technique such as the Alternating Direction Implicit (ADI) method [14]. This technique assumes that the five star stencil operator $\mathcal{L}$ for the finite difference scheme can be split into two line operators:

$$\mathcal{L}_x\psi = -\psi_{i+1,j} + 2\psi_{i,j} - \psi_{i-1,j} \quad (9)$$

$$\mathcal{L}_y\psi = -\psi_{i,j+1} + 2\psi_{i,j} - \psi_{i,j-1} \quad (10)$$

which are invoked alternatively at each half-time step. This formalism effectively 'decouples' the data dependency in the orthogonal direction to each of the two operators $\mathcal{L}_x$ and $\mathcal{L}_y$, to allow the computation to be carried out in parallel in the orthogonal directions.

The code kernel in appendix A.8.1 illustrates this technique. For simplicity this code solves only the time-independent equation (8), but since the essence of the ADI method is to solve each one of the operator equations (9) & (10) alternatively at each half-time step, a fictitious time variable $\hat{t}$ was introduced, such that the equation to solve became:

$$\frac{\partial\psi}{\partial\hat{t}} = \nabla^2\psi - \zeta, \quad (11)$$

to be iterated until the equilibrium solution was found. Associated routines are also listed for solving the resulting tridiagonal matrix (appendix A.8.2) and for calculating the right hand side to solve for (appendix A.8.3), at each half-time step.

The exchange between the two ADI directions mean that some form of data redistribution has to be performed between each half-time step in order for the parallelism in the computation to be maintained. In HPF, the most obvious solution would be to declare the relevant distributed arrays with the '`DYNAMIC`' attribute, and then call the '`REDISTRIBUTE`' directive to alternate between the two orthogonal distribution directions. For example:

```
      REAL,DIMENSION(N,N) :: psi,zeta
!HPF$ DYNAMIC :: psi,zeta
!HPF$ DISTRIBUTE (*,BLOCK) :: psi,zeta
      ...
      DO WHILE( .NOT. converged )
         CALL tridiag_psi_x(zeta,psi)
         CALL find_rhs_psi_x(zeta,psi)
!HPF$    REDISTRIBUTE (BLOCK,*) :: psi,zeta
         CALL tridiag_psi_y(zeta,psi)
         CALL find_rhs_psi_y(zeta,psi)
!HPF$    REDISTRIBUTE (*,BLOCK) :: psi,zeta
      END DO
```

Alternatively, the approach in appendix A.8.1 can be used, which calls the Fortran 90 '`TRANSPOSE()`' intrinsic function on the same distributed arrays to make the exchange between the two spatial directions. At the time of writing, the use of the '`TRANSPOSE()`' intrinsic function is better supported on the current crop of HPF compilers, and has the added bonus of being within the Subset HPF specifications. As this matrix remapping/transpose contains the costliest part of the communications, an effective parallelization of ADI-like applications would depend on the compiler's implementation of this step.

6.6.2 Panel Methods

Panel methods are widely used in the aerospace and automotive industry and are effectively boundary-element methods for computational fluid dynamics problems. These methods employ the surface of the body over which fluid is flowing to be used as the computational domain rather than using the whole region in which the body is embedded. This is not only computationally more efficient than a finite difference method, for example, but also allows more complicated body shapes to be studied than would be tractable if the body were embedded in a regular mesh.

The computational structure required for solving the above set of equations and is the same as Gaussian elimination:

- converting the matrix into a diagonal form where the lower part is zero. Here the row used to eliminate lower diagonal parts is chosen to improve numerical stability, a process known as pivoting.
- forward elimination where the transformations and multiplications performed on the matrix are repeated on the RHS vector.
- Removing the above diagonal terms by back substitution.

The use of HPF for Gaussian Elimination is described in §6.1.

6.7 Particle Dynamics

An important problem in astrophysics and molecular dynamics is that of N-body interactions, where a number of particles or bodies, interact and some dynamical scheme such as Newton's laws of motion is used to integrate the equations of motion of the particles under the given interaction force. Typically some empirical model such as the inverse-square force law of gravity or some other potential model of interaction is used to specify how any given pair of particles interact. The principle of superposition is used to combine pair-wise forces to produce the total force or potential on a given particle. Generally it is the calculation of this total force that is the most computationally intensive. Thereafter the motion equations are usually updated by a time stepping algorithm that is embarrassingly data parallel as it involves exactly the same operations for each particle.

6.7.1 Direct Methods

These method is useful when the number of particles to be be simulated are small. They involve calculation of all $O(N^2)$ interactions between the particles. An example of such a calculation is vortex methods in hydrodynamics. For small number of particles they outperform other

methods. The crossover point where using other techniques are better than this approach is of order of 10,000 particles.

A simple version of this method can be implemented by using a simple do-loop which shifts the array representing the bodies by one step in each of its N iterations. The '`CSHIFT()`' communications intrinsics function can be used to rotate data for the j'th particle past the processor holding data for the i'th particle. Thus the i–j'th contributions can be built up in turn, as is illustrated in Figure 4. When the pairwise interactions are commutative, for example as would be found in classical molecular dynamics and Newtonian gravitational interactions, the total amount of computation can be reduced in half by calculating the interaction only once and using Newton's third law of motion to automatically determine the other half [29]. This requires a simple modification of the N-Body code, with the main complication being an explicit check on whether an even or an odd number of particles are being considered. The code in appendix A.9 is such an example of a commutative N-Body direct simulation. It calculates the forces between a given number of particles at a single time step, for a system of gravitationally interacting particles.

However, the code given is not expected to generate good performance on distributed memory machines as the number of messages generated is large: essentially, a message is sent for each of the (roughly) $\frac{N}{2}$ '`CSHIFT()`' operations. The resulting setup overhead of these messages would dominate the total communication cost on most machines. An intelligent compilers can perhaps combine several of these '`CSHIFT()`' operations together to reduce this overhead. Alternatively, one can re-express the algorithm by making use of the block distribution of the data arrays. The number of '`CSHIFT()`' operations would then be decreased from $\frac{N}{2}$ to $\frac{N}{2p}$, where p is the number of participating processors.

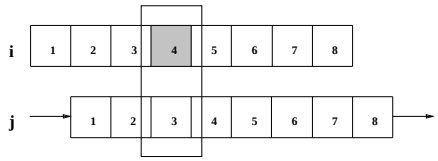


Figure 4: Use of successive CSHIFT operations to rotate vector of particle separations past each element in turn.

6.8 Image and Signal Processing

Image processing is the computationally intensive part of several applications in scientific computing, data compression, computer vision and robotics. Images are typically stored as two or three dimensional arrays. Many of the image processing kernels have are highly data parallel and can be effectively represented using High Performance Fortran.

6.8.1 Windowing Applications

These applications require performing simple operations at every point on the image using all the pixels in a small window (typically 3×3 to 5×5). These can be represented by a template matching operation which requires convolving the image, I (of size say $N \times N$), with a two dimensional template, T (of size say $M \times M$), using the following:

$$C2D[i,j] = \sum_{u=0}^{M-1} \sum_{v=0}^{M-1} I[(i+u) mod N, (j+v) mod N] * T(u,v)$$

A high value of $C2D(i,j)$ implies that the $M \times M$ subimage at (i,j) is similar to the template. Different templates can be used to perform operations such as edge detection and mean filtering. Similar windowing operations are also required for calculating the effective shift for MPEG based data compression.

These calculations can be effectively represented in High Performance Fortran using '`FORALL`' loop or '`CSHIFT`' intrinsic. The image is distributed among the processors using '`BLOCK`' distribution along one or more dimensions. The code fragment for template matching is given in appendix A.10.

The computation cost per pixel these operations is limited. Careful mapping of communication is required for high performance and scalability because the ratio of computation/communication required is small. Limited performance may be achieved for small images for architectures with high startup latency costs.

6.8.2 Histogramming Applications

Filtering operation based on histogram equalization require calculation of the number of pixels of different gray levels (number of levels vary from 2 to 256). This can be achieved using '`*_SCATTER()`' / '`*_GATHER()`' intrinsics. Such an application is very similar to that already discussed in §5.2 for the Hough transformation.

In HPF, the key computation in building up the histogram would be done with the intrinsic function '`COUNT_SCATTER()`', as illustrated in the following code fragment, for an image of N pixels by N pixels of 1–P possible gray levels:

```
      INTEGER,DIMENSION(N,N) :: image,ind2
      INTEGER,DIMENSION(P,N) :: distHistogram
      INTEGER,DIMENSION(P) :: histogram
!HPF$ DISTRIBUTE (*,BLOCK) :: image,ind2
!HPF$ DISTRIBUTE (*,BLOCK) :: histogram
      FORALL( k=1:N ) ind2(:,k) = k
      countMask = .TRUE.
      countBase = 0
      distHistogram = COUNT_SCATTER( countMask, countBase, image, ind2 )
      histogram = SUM( distHistogram,DIM=2 )
```

where a temporary 2-dimensional array '`distHistogram`' is used to exploit the parallelism in the code, by distributing the counts over the pixel values amongst the available processors.

These counts are then summed into the final histogram along the distributed rank-2 direction by another intrinsic function. The other workspace array '`ind2`' are the rank-2 indices, set so as to keep references to the '`distHistogram`' array to be local to each processor.

6.8.3 Image Transformations

Image Transformations include operations such as image translation, rotation, and scaling. Translation can be represented by a simple '`FORALL`' loop or an array assignment statement. Rotation can be represented using a '`FORALL`' loop. However, it involves highly unstructured as well as a large amount of data movement (unless the amount of rotation is small) and may requires careful mapping of data movement to the underlying communication network for achieving good performance.

6.8.4 Fast Fourier Transform

The Fast Fourier Transform (FFT) is the most widely known example of the Spectral method for computational problems. In all such methods, one performs a linear transformation of the stated problem into another physical domain where it is hoped will be more tractable. In Fourier transformations, the mapping is from the time-domain to the frequency-domain. This pattern appears in diverse applications such as medical imaging and synthetic aperture radar imaging.

An example code (appendix A.11.1) performs a two-dimensional FFT in the following manner:

1. Perform FFT across the leading dimension of the matrix.
2. Transform the matrix, using the F90/HPF '`TRANSPOSE()`' intrinsic function.
3. Perform FFT again, across the new leading dimension.

The 2-D matrix is distributed in a one-dimensional manner, over the second dimension of the 2-D matrix. This allows a 1-D FFT (appendix A.11.2 across the memory-contiguous leading dimension index to be performed in parallel.

The basic mechanism to allow the computation to be parallelized is the HPF '`INDEPENDENT`' directive. The called subroutine is given a slice of the array '`a()`', and performs each 1-D FFT in parallel with the other columns. The quantity '`sign`' is a flag to indicate whether an FFT, or the inverse operation should be performed.

A simple way to calculate 1-D FFT is similar to the above calculation of 2-D FFT. An $n = n_1 n_2$ FFT is computed as an $n_1 \times n_2$ array A, stored in column major order. This is followed by an application of 2D-FFT algorithm described above. A simple implementation requires three redistributions. However, this can be reduced to only one redistribution by using a different algorithm. It requires changing the view from '`BLOCK`' to '`CYCLIC`' distribution

locally and can only be achieved by an explicit reordering by data copying in the current HPF standard.

Another way to implement the 1-D FFT is a direct parallelization of radix 2 FFT algorithm [27]. A representative code is given in appendix A.12. Most of the communication is generated in the last phase of the program. It can be shown that the scalability of this algorithm is than the transpose based FFT due to asymptotically larger amount of communication required [27].

The FFT algorithm can be used to compute 2-D convolution (appendix A.13). Let two equal-sized images '`A`' and '`B`' are represented in a (complex) matrix of square dimension '`NN`'. The convolution of these images are stored in the matrix '`C`', and is calculated by the steps:

1. Perform 2-D FFT for Image '`A`'.
2. Perform 2-D FFT for Image '`B`'.
3. Perform matrix multiplication '`C`' = '`FFT(A) * FFT(B)`'.
4. Perform inverse 2-D FFT on '`C`' to obtain the convolution. where the Fast Fourier Transforms (FFT) are calculated using the method described in the HPFA 2-D FFT kernel.

6.9 Neural Networks

6.9.1 Hopfield Networks

These networks have been used as models for auto-associative memory, as is conjectured to be found in the hippocampus region of the brain, and for optimization tasks ([23]). For simplicity, we assume that input (and output) patterns are binary vectors, and each of the N nodes' state is $+1$ or -1. The activation functions of nodes are step functions, applied to the sum of external inputs and the net weighted inputs from other nodes. Node output values may change when an input vector is presented, until the state of the network converges to an attractor. The update dynamics of each site from time t to $t+1$ is given by the function:

$$S_i(t+1) = \mathrm{sgn}\left(\sum_{j=1}^{N} w_{i,j} S_j(t) + I_i\right) \tag{12}$$

where $\mathrm{sgn}(x) = 1$ if $x \geq 0$ and $\mathrm{sgn}(x) = -1$ if $x < 0$. According to the traditional definition of the sigma function, $\mathrm{sgn}(0) = 1$. The term $\vec{I}$ is the external input vector, and can be used to bias the network with a persistent guess of the image to be retrieved ([42]).

The Hopfield prescription for the setting of the network's weights w_{ij} are derived from the biologically based Hebbian ([17]) learning process. Namely as a sum over the $\mu = 1 \cdots P$ patterns the network is asked to store:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^{P} \xi_i^\mu \xi_j^\mu \tag{13}$$

which is of course the outer product of each of these patterns. Since this weight matrix is the square of the number of nodes (and the number of image pixels), it dominates the memory requirement of a Hopfield network code which HPF's paradigm of distributing arrays across different processors would alleviate. Note that the Hebbian training phase may represent a substantial contribution to the overall execution time, as it is difficult to parallelize without replicating each pattern $\vec{\xi}^{\mu}$ over the processors —which may be prohibitively expensive if large number of nodes and patterns are to be stored in the simulation for measuring the 'performance' of the network via the overlap measure:

$$m^{\mu}(t) = \frac{1}{N} \sum_{i=1}^{N} S_i(t) \xi_i^{\mu}. \tag{14}$$

If all the nodes are updated synchronously, the update can be easily completed by matrix vector multiply distributing the weight matrix using a '`BLOCK`' or '`CYCLIC`' distribution and aligning the solution vector with the first row of the weight matrix. However, an important assumption in the traditional Hopfield model is updates are asynchronous: *i.e.*, at every time instant, precisely one node's output value is updated. The selection of a node for updating may be random. Very often the selection is made in a cyclic manner, updating one node after another. This would reduce the amount of available parallelism significantly. This can be approximated by generating a random vector which only updates a subset of all the nodes in a given iteration. This would require using a '`CYCLIC`' distribution of the weight array to maintain good load balance. The convergence properties of such an algorithm is not well defined. A representative Hopfield code, including the storage and retrieval phases, is given in appendix A.14.

6.9.2 Learning Vector Quantizers

Unsupervised learning and clustering can be useful preprocessing steps for solving classification problems. A learning vector quantizer ([26]) is an application of winner-take-all networks for such tasks, and illustrates how an unsupervised learning mechanism can be adapted to solve supervised learning tasks in which class membership is known for every training pattern.

Each node in an LVQ is associated with an arbitrary class label. The number of nodes chosen for each class is roughly proportional to the number of training patterns that belong to that class, making the assumption that each cluster has roughly the same number of patterns. The update rule in an LVQ is slightly different from that of the competitive learning rule discussed earlier for winner-take-all networks:

> Let pattern $\boldsymbol{x}$ from class $C(i)$ is presented to the network, and the winner node $j*$ belongs to class $C(j)$. The winner is moved either towards or away from the pattern depending on whether $C(i) = C(j*)$.

The algorithm is presented in figure 5 [32]. The learning rate $\eta(t)$ is a decreasing function of time t, such as $\eta(t) = \eta_0 t^{-A}$ or $\eta(t) = \eta_0[1 - At]$ where η_0 and A are positive constants. This allows the network to 'converge' to a state in which the weight vectors are stable and change little with further input sample presentations.

Algorithm LVQ1();

Initialize all weights to some random values in the range [0,1].
repeat

for each input pattern $\boldsymbol{x}_k$ in the training set **do**
find the node $j*$ whose weight vector $\boldsymbol{w}_{j*}$ is closest to $\boldsymbol{x}_k$.
for each node j in the current neighborhood of $j*$ **do**
/* change the weight vector $\boldsymbol{w}_j$ by an amount $\Delta\boldsymbol{w}_j$, */
for $i = 1, \ldots, n$ **do**
$\Delta w_{ji} = \eta(t)(x_{ki} - w_{ji})$ if the class label of
node j = the desired class of $\boldsymbol{x}_k$,
$\Delta w_{ji} = -\eta(t)(x_{ki} - w_{ji})$ otherwise.

Adjust the correction factor $\eta(t)$.

until the network has converged or computational bounds are exceeded

Figure 5: LVQ1 algorithm

6.10 Evolutionary Computation

Evolutionary algorithms are stochastic state-space search techniques modeled on natural evolutionary mechanisms. These methods have been successful in solving many difficult optimization problems. However, they are computationally very intensive and can effectively use parallel machines.

They maintain a population of feasible solutions, and attempt to improve the overall quality of these populations as time progresses. Each individual in the population represents one potential solution for the optimization problem. Evolutionary computing methods follow roughly three lines, among which distinctions become increasingly blurred.

1. *Evolutionary Programming (EP)* emphasizes direct representations of individuals, and mutation as the primary mechanism for generating new individuals.

2. *Evolutionary Strategies* use real-valued genes and self-adapting strategy variables (such as covariances) in the search process.

3. *Genetic Algorithms* traditionally manipulate the genetic makeup of individuals, using fixed-length binary string representations, and 'crossover' operators which combine gene segments of parent individuals to generate offspring.

Figure 6.10 describes a generic evolutionary computation algorithm [32]. $\mathcal{P}(t)$ denotes the population of size N at time $t \geq 0$. $O_1, \cdots, O_\ell$ are operators that can be applied to sets of individuals to generate other individuals. The quality of an individual is gauged by a *fitness function* f. Members of $\mathcal{P}(0)$, the initial population, are often chosen randomly; this is always the case in evolutionary programming and strategies.

Procedure Evolutionary Computation:
  Initialize the population $\mathcal{P}(0)$;
  while *termination criterion* is not satisfied, **do**
    Reproduction: Obtain subsets C_i of individuals from the
      current population $\mathcal{P}(t)$
      (where $\mid C_i \mid = 1$ for mutation and 2 for crossover);
    Operator Application: To each C_i, apply operators
      that result in new individuals,
      and let $\Omega(t)$ be the resulting offspring set;
    Selection: Obtain $\mathcal{P}(t+1)$ by selecting $\mid \mathcal{P}(t) \mid$
      individuals from $\mathcal{P}(t) \cup \Omega(t)$;
  end-while
  return the best individual from the current population.

Figure 6: A generic evolutionary computation algorithm

Parallelism in each step of the evolutionary computation algorithm can be represented using High Performance Fortran. The reproduction step requires performing an '`INDEPENDENT`' loop or a '`FORALL`' loop. The operator application can be completed using an independent do loop. Typically the fitness evaluation of the offspring is the most computationally intensive part of the whole algorithm. The selection operation requires sorting (using '`GRADE`' operation) of all the offsprings generated based on their fitness. This is followed by selecting the offsprings with the highest fitness (there is a small amount of randomness involved) followed by a '`PACK`' operation.

6.11 Graph Algorithms

Many applications can be expressed and solved using graphs. In this section we discuss the usefulness of HPF for expressing many of these algorithms. The algorithm described below are asymptotically and practically efficient only when the input graph is dense (*i.e.*, most of the edges are present). Graphs for a large number of real applications tend to be sparse. The algorithms for such graphs are difficult to parallelize and require efficient implementation of data structures such as priority queues. Many of these algorithms are difficult to represent with HPF.

6.11.1 All-pairs shortest path algorithm

This problem involves finding the shortest path between all vertices $v_i, v_j, i \leq j$ for a weighted graph $G(V, E)$. Let $w(v_i, v_j)$ represent the weight between vertex v_i and v_j. There are three parallel methods which can be used for the solution of this problem:

1. Assuming the number of nodes in the graph is n, the single source shortest path algorithm can be applied n times using the '`INDEPENDENT`' directive. A good implementation of this would require maintaining a copy of the whole graph on every processor. This can be achieved by using replication of the graph array but may high memory requirements. This approach is limited to small sized graphs.

2. A dynamic programming-based approach can be used to calculate the shortest paths using the following problem formulation:

   Let $d(i,j)$ represent the shortest distance between i and j such that the largest intermediate vertex is less than or equal to k.

   $$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & k = 0 \\ \min\limits_{k} \left\{ d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{i,j}^{(k-1)} \right\} & k \geq 1 \end{cases}$$

   where $w(v_i, v_j)$ is the weight of the edge $\mathrm{S}(v_i, v_j)$, and $d_{i,j}^{(n)}$ represents the desired solution.

   This can be computed using the following algorithm

   $d_{i,j}^{0} = w\ v_i, v_j.$
   do $k = 1$ to n do
   forall $i = 1$ to n do
   forall $j = 1$ to n do
   $$d_{i,j}^{(k)} = \min \left\{ d_{i,j}, d_{ij,k}^{k-1} + d_{k,j}^{k-1} \right\}$$

   Minimizing the Communication requires a '`(BLOCK, BLOCK)`' distribution of d array.

3. The third algorithm is based on matrix multiplication. It can be shown that the all-pairs shortest path can be solved using $\log n$ iterations of matrix multiplication. This can be easily code in HPF using the '`MATMULT()`' intrinsic. This algorithm is work inefficient and would not perform well unless the number of processors is much larger than the size of the matrix.

6.11.2 Minimum spanning tree

A minimum spanning tree of the graph is a spanning tree (a tree which contains all the vertices) with minimum weight. An algorithm for calculating the minimum spanning assumes that set S initially contains an arbitrary vertex r [6, 27]. The initial weight of the remaining vertices is given by the cost of the edge (if any) to the vertex r. The following steps are executed $n-1$ times (i.e. till the set S contains all the vertices):

Step 1: Find the vertex with minimum weight to any vertex in set S.

Step 2: Move the vertex to set S.

Step 3: Update the weight of the remaining vertices based on a potentially new shortest distance via the newly moved vertex.

This algorithm can be implemented using a mask array of size n. The mask of vertex i is set to 0 if vertex $i \in S$. Initially the mask of all vertices except r is set to 0. Step 1 can be completed using the computational intrinsic '`MIN()`' in HPF. Step 2 updates the mask array using an assignment statement. Step 3 can be completed using a forall loop.

The above algorithm representation requires checking the mask vector of all the elements participating at each step. Once the mask value of a vertex changes, it does not participate in Step 1 or Step 3. This check can potentially be performed using a '`WHERE`' construct. Good load balancing can typically be achieved by using a '`CYCLIC`' distribution.

6.11.3 Single Source Shortest Path

This problem requires calculating the shortest path from a given vertex to all the remaining vertices in the graph. The single source shortest path algorithm for dense graphs is a variant of of the minimum spanning tree algorithm described above [6]. The parallelization requirements are also similar [27].

6.12 Summary of Synchronous Applications

Computational problems where the algorithm is synchronous, *i.e.*, data parallelism with regular data and identical operations on all data points, are easily expressable in a parallel manner with High Performance Fortran. This is illustrated in this section with the algorithms discussed above, and summarised in table 2.

In these applications, data mappings are largely static and can be described with the '`BLOCK`' or '`CYCLIC`' distribution formats. Where remappings are required, they are simple enough to be expressable via the '`TRANSPOSE)`' intrinsic function. Other communication demands are via the use of the '`CHSIFT()`' function, reduction operators such as '`MAXVAL()`', and matrix & vector multiplication functions. The parallelism in the algorithms can be expressed using the language constructs (*i.e.*, '`FORALL`'and '`WHERE`'), Fortran 90 array syntaxes and the calling of (parallelised) intrinsic functions. The efficient implementation of these features can probably be regarded as a minimum requirement for those wishing to express their synchronous application in HPF. Our experience [43] is that these requirements are being seriously addressed with today's available compilers.

Application	HPF Features
Gaussian Elimination (A.4)	'CYCLIC' distribution, 'FORALL', 'MAXLOC()'
2-D Potts model (A.6)	'CSHIFT()', 'MERGE()', 'WHERE'
2-D Cahn-Hilliard (A.7)	'CSHIFT()'
Poisson solution by ADI (A.8)	Descriptive interface, 'TRANSPOSE()'
Direct N-body (A.9)	'CSHIFT()'
2-D FFT & Convolution (A.11,A.13)	Array syntax, 'INDEPENDENT', 'TRANSPOSE()'
Hopfield Neural Network (A.14)	'WHERE', 'MATMUL()', 'RESHAPE()', 'MAXVAL()', 'DOT_PRODUCT()', 'RANDOM_NUMBER()'

Table 2: Synchronous applications discussed in this paper with the appendices where their HPF implementations can be found, and the important F90/HPF features which they exploit. Intrinsic functions are differentiated from keywords by the '()' postfix. The use of array syntax is common to all these codes and so have been left out for brevity.

7 Conclusions

The intent of this paper has been as a motivation to programmers to consider the High Performance Fortran language as a means of exploiting the data parallelism in their problems. This has been done by first explaining the key features of HPF: how it builds on the Fortran 90 standard, and how its single-threaded execution model allows the user to run the same code for one or for many processors. That is, and in contrast to the traditional parallelisation methodology of inserting calls to message passing subroutines, the changes made to the code in the name of HPF may be considered essentially benign[3].

We have also seen in this paper a range of applications for which the High Performance Fortran language has shown itself to be suitable. From these applications we hope to have either provided concrete examples on how they may be programmed in HPF, or have provided fragments demonstrating the pertinent language features. These have included applications in magnetic spin models, computational fluid dynamics, financial simulations, image processing, particle dynamics, and the solution of matrix equations. The codes are part of the NPAC HPF Applications suite, and the further expanding and benchmarking them is an ongoing process. But we can already state that for many of the codes in this paper, we have seen good scalability up to eight processors.

In this study we have restricted ourselves to synchronous or embarrassingly parallel problems, a constraint brought on by the lack of available application codes and by the capabilities of the current crop of HPF 1.1 compilers. With the recent publication of the HPF 2.0 standard, it is hoped that compilers will shortly be expanded so as to allow the HPF language to

[3]Subject to the language differences between Fortran 90 and HPF outlined in §3.2.2–3.2.4.

tackle the more general irregular case. This new standard also offers the possibility of allowing the programmer to express the code in a task-parallel manner, in lieu of the current data-parallel model. In theory, this offers a better opportunity to explore parallelism without a prior conversion to the array syntax of Fortran 90, as is the current case with HPF 1.1.

Acknowledgements

The authors wish thanks for advice and writing of some of the HPF kernel codes covered in this paper should be extended to the following colleagues: Bryan Carpenter, Alok Choudhary, Paul Coddington, Gang Cheng, Tom Haupt, Miloje Makivić and Don Leskiw. In addtion, work by the following graduate students were appreciated: Mike McMahon, Rajeev Thakush, Rajesh Bordawekar, Chao-Wei Ou, Jhy-Chun Wang and Kivanc Dincer.

This research is supported in part by NSF under ASC-9213821 and AFMC and ARPA under contract #F19628-94-C-0057. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

A Example Codes

The following subsections are codes written as part of the NPAC HPF Application Kernel suite, and represent instances of the applications and HPF features described in earlier in this paper. Those codes which were originally written in Fortran 77 usually required extensive rewriting; but those ported from existing data parallel languages such as Thinking Machines' CM-Fortran, or MasPar's Fortran were more straightforward. Similarly, a code written in Fortran 90 would fall somewhere between these two cases.

The latest versions of these codes may be found in the URL:

`http://www.npac.syr.edu/hpfa/kernels/number.html`

A.1 NAS EP Benchmark - Tabulation of Random Numbers

This is the NPAC version of the NAS Embarrassingly Parallel benchmarks code, and generates random numbers with a Gaussian probability distribution, based on the Box-Muller algorithm.

```
      PROGRAM ep
      IMPLICIT NONE
      INTEGER, PARAMETER          :: M = 20, MK = 4
      INTEGER, PARAMETER          :: MM = M-MK
      INTEGER, PARAMETER          :: NK = 2**MK
      INTEGER, PARAMETER          :: NN = 2**MM
      INTEGER, PARAMETER          :: NQ = 10
C M  = log2 number of [0,1] uniform random pairs to create.
C MK = log2 size of each batch of [0,1] uniform pairs to create.
C      Where NK = 2**MK.
C MM = log2 number of independent batches to create, divisible with
C      the number of available processors.
C      Where NN = 2**MM.
C NQ = Number of bins with which to gather statistics on the Gaussian
C      deviates.
      DOUBLE PRECISION, DIMENSION(NN) :: X,Y
      DOUBLE PRECISION, DIMENSION(NN) :: X1,X2,T2,T3,T4,GC
      INTEGER, DIMENSION(NN)          :: KK
      INTEGER, DIMENSION(0:NQ,NN) :: QQ
      INTEGER, DIMENSION(0:NQ)    :: Q
      INTEGER I,J,NGCI
      INTEGER iStep
      DOUBLE PRECISION NGC
      DOUBLE PRECISION, PARAMETER :: A = 1220703125.D0
      DOUBLE PRECISION, PARAMETER :: S =  271828183.D0
      REAL :: ti,tf
      REAL :: tstart,tend
      REAL :: tcomms, tcomp
      REAL :: randComp
!HPF$ PROCESSORS procs( NUMBER_OF_PROCESSORS() )
!HPF$ TEMPLATE TTT(NN)
!HPF$ DISTRIBUTE TTT(BLOCK) ONTO procs
!HPF$ ALIGN (:) WITH TTT(:) :: X,Y,X1,X2,T2,T3,T4,GC,KK
!HPF$ ALIGN QQ(*,:) WITH TTT(:)
      INTERFACE
        SUBROUTINE random_seed(s, a, x, MK, NN)
```

```
        DOUBLE PRECISION, DIMENSION(:), INTENT(OUT) :: x
!HPF$   DISTRIBUTE *(BLOCK) :: x
        DOUBLE PRECISION, INTENT(IN)             :: a,s
        INTEGER, INTENT(IN)                      :: MK,NN
        END SUBROUTINE RANDOM_SEED
        SUBROUTINE randlc (x, a, y, NN, myComp)
        DOUBLE PRECISION, DIMENSION(:), INTENT(INOUT)  :: x
        DOUBLE PRECISION, DIMENSION(:), INTENT(OUT)    :: y
        DOUBLE PRECISION, INTENT(IN)               :: a
        INTEGER, INTENT(IN)                        :: NN
        REAL, INTENT(INOUT)                        :: myComp
!HPF$   DISTRIBUTE *(BLOCK)  :: x,y
        END SUBROUTINE randlc
      END INTERFACE
      INTERFACE
         SUBROUTINE timer(return_time, initial_time)
         REAL, INTENT(IN)  :: initial_time
         REAL, INTENT(OUT) :: return_time
         END SUBROUTINE timer
      END INTERFACE
C Begin Executable
      tcomms = 0.0
      tcomp  = 0.0
      randComp = 0.0
      iStep = NK / 10
      IF( iStep .EQ. 0 ) iStep = 1
      X  = 0.0
      Y  = 0.0
      X1 = 0.0
      X2 = 0.0
      T2 = 0.0
      T3 = 0.0
      T4 = 0.0
      GC = 0.0
      KK = 0
      QQ = 0
      Q  = 0
C start_timer
      CALL timer(ti, 0.0)
      CALL random_seed(S,A,X,MK,NN)
      QQ=0
      DO I = 1, NK
         IF( MOD(i,iStep) .EQ. 0 )THEN
            WRITE(UNIT=*,FMT='(''Status i = '',I8)') i
         END IF
C Compute Gaussian deviates by acceptance-rejection method and
C tally counts in concentric square annuli.
        CALL randlc (x, a, y, NN, randComp)
        CALL timer(tstart,0.0)
        X1 = 2.D0 * Y - 1.D0
        CALL timer(tend,tstart)
        tcomp = tcomp + tend
        CALL randlc (x, a, y, NN, randComp)
        CALL timer(tstart,0.0)
        X2 = 2.D0 * Y - 1.D0
        Y = X1 ** 2 + X2 ** 2
        WHERE (Y .LE. 1.D0)
          GC = GC + 1.D0
          T2 = SQRT (-2.D0 * LOG (Y) / Y)
          T3 = ABS (X1 * T2)
```

```
          T4 = ABS (X2 * T2)
          KK = MAX (T3, T4)
        ELSEWHERE
          KK = NQ
          Y = -10.0D0
        ENDWHERE
        DO J = 0, NQ - 1
          WHERE (KK .EQ. J) QQ(J,:) = QQ(J,:) + 1
        ENDDO
        CALL timer(tend,tstart)
        tcomp = tcomp + tend
      ENDDO
      CALL timer(tstart,0.0)
      Q   = SUM(QQ,DIM=2)
      NGC = SUM(GC)
      CALL timer(tend,tstart)
      tcomms = tcomms + tend
C Stop timer.
      CALL timer(tf,ti)
      tcomp = tcomp + randComp
C This is done so the WRITE statement works
      NGCI = INT(NGC)
      PRINT *,'time taken is ',tf,'seconds'
      WRITE(6,1001) M,NGCI,(I,Q(I),I=0,NQ-1)
      WRITE(UNIT=*,FMT=6004) NUMBER_OF_PROCESSORS(),NN,
     1                       tcomms,tcomp,(tf-tcomms-tcomp),tf
      STOP
 1001 FORMAT('BENCHMARK 1 RESULTS:' // 'N = 2^',I8/
     &   'NO. OF GAUSSIAN PAIRS =',I15/'COUNTS:'/(I3,I15))
 2001 FORMAT(12E12.3)
 6004 FORMAT('Number of Processors = ',I4/
     1       'Problem size = ',I8/
     2       'Communications = ',F9.3/
     3       'Compute        = ',F9.3/
     4       'Others         = ',F9.3/
     5       'Total time     = ',F9.3)
      END
C======================================================================
      SUBROUTINE random_seed(s, a, x, MK,NN)
C ----------------------------------------------------------------------
C|                                                                     |
C| Routine to generate array x(NN) of seeds for parallel random number |
C| generator.                                                          |
C|                                                                     |
C| As with RANDLC, compute R23 = 2 ^ -23, R46 = 2 ^ -46,               |
C| T23 = 2 ^ 23, and T46 = 2 ^ 46.  These are computed in loops, rather|
C| than by merely using the ** operator, in order to insure that the   |
C| results are exact on all systems.  This code assumes that 0.5D0 is  |
C| represented exactly.                                                |
C|                                                                     |
C| H W Yau.  15th of August, 1996.                                     |
C| Northeast Parallel Architectures Center.                            |
C| Syracuse University.                                                |
C|---------------------------------------------------------------------|
C| Edit record:                                                        |
C| 15/Aug/1996: Corrected comments.                                    |
C|              Corrected generation of the seeds x(NN).               |
C|---------------------------------------------------------------------|
      IMPLICIT NONE
      DOUBLE PRECISION, DIMENSION(:), INTENT(OUT) :: x
```

```
      DOUBLE PRECISION, INTENT(IN)             :: a, s
      INTEGER, INTENT(IN)                      :: MK,NN
!HPF$ DISTRIBUTE *(BLOCK) :: x
      DOUBLE PRECISION :: t1,t2,zz1,zz2,z
      DOUBLE PRECISION R23, R46, T23, T46, an, an1, an2
      DOUBLE PRECISION, DIMENSION(NN) :: zz
      INTEGER I
C Begin Executable
      R23 = 1.D0
      R46 = 1.D0
      T23 = 1.D0
      T46 = 1.D0
      DO  I = 1, 23
         R23 = 0.5D0 * R23
         T23 = 2.0D0 * T23
      ENDDO
      DO I = 1, 46
         R46 = 0.5D0 * R46
         T46 = 2.0D0 * T46
      ENDDO
C Calculate: an = a^MK mod 2^46.
      an = a
      DO i = 1, MK+1
C Perform (an * an) MOD 2^46.  First, split 'an' into two parts such that
C an = 2^23 * an1 + an2.
         an1 = AINT( R23 * an )
         an2 = an - T23 * an1
C Compute z = an1*an2 + an2*an1 (mod 2^23), and then
C an = 2^23 * z + an2*an2 (mod 2^46).
         t1 = an1 * an2 + an2 * an1
         z  = t1 - T23 * AINT( R23 * t1 )
         t2 = T23 * z + an2 * an2
         an = t2 - T46 * AINT( R46 * t2 )
      END DO
C Now compute the seeds, into a non-distributed array zz().  First,
C split 'an' into two parts such that
C an = 2^23 * an1 + an2.
      an1 = AINT( R23 * an )
      an2 = an - T23 * an1
      zz(1) = s
      DO i = 2,NN
C Break z(i-1) into two parts, such that z(i-1) = 2^23 * zz1 + zz2.
         zz1 = AINT( R23 * zz(i-1) )
         zz2 = zz(i-1) - T23 * zz1
C Compute z = an1*zz2 + an2*zz2 (mod 2^23), and then
C     zz(i) = 2^23*z + an2*zz2 (mod 2^46).
         t1    = an1 * zz2 + an2 * zz1
         z     = t1 - T23 * AINT( R23 * t1 )
         t2    = T23 * z + an2 * zz2
         zz(i) = t2 - T46 * AINT( R46 * t2 )
      END DO
C Copy into distributed array.
      x = zz
      RETURN
      END
C=====================================================================
      SUBROUTINE RANDLC (X, A, Y, NN, myComp)
      IMPLICIT NONE
      DOUBLE PRECISION, DIMENSION(:), INTENT(INOUT)  :: X
      DOUBLE PRECISION, DIMENSION(:), INTENT(OUT)    :: Y
```

```
      DOUBLE PRECISION, INTENT(IN):: A
      INTEGER, INTENT(IN)         :: NN
      REAL, INTENT(INOUT)         :: myComp
      DOUBLE PRECISION, DIMENSION(SIZE(X)) :: A1,A2,T1,T2,T3,T4,X1,X2,Z
      DOUBLE PRECISION R23, R46, T23, T46
      REAL :: tstart, tend
      INTEGER I,KS
      SAVE KS, R23, R46, T23, T46
      DATA KS /0/
!HPF$ DISTRIBUTE  *(BLOCK) :: x,y
!HPF$ ALIGN (:) WITH x(:) :: A1,A2,T1,T2,T3,T4,X1,X2,Z
C A is constant
C given array X(NN) of seeds
C generate array Y(NN) of random numbers
C If this is the first call to RANDLC, compute R23 = 2 ^ -23, R46 = 2 ^ -46,
C T23 = 2 ^ 23, and T46 = 2 ^ 46.  These are computed in loops, rather than
C by merely using the ** operator, in order to insure that the results are
C exact on all systems.  This code assumes that 0.5D0 is represented exactly.
C Begin Executable
      CALL timer(tstart,0.0)
      IF (KS .EQ. 0) THEN
         R23 = 1.D0
         R46 = 1.D0
         T23 = 1.D0
         T46 = 1.D0
         DO  I = 1, 23
            R23 = 0.5D0 * R23
            T23 = 2.D0 * T23
         ENDDO
         DO  I = 1, 46
            R46 = 0.5D0 * R46
            T46 = 2.D0 * T46
         ENDDO
         KS = 1
      ENDIF
C Break A into two parts such that A = 2^23 * A1 + A2.
      T1 = R23 * A
      A1 = AINT (T1)
      A2 = A - T23 * A1
C Break X into two parts such that X = 2^23 * X1 + X2, compute
C Z = A1 * X2 + A2 * X1  (mod 2^23, and then
C X = 2^23 * Z + A2 * X2  (mod 2^46).
      T1 = R23 * X
      X1 = AINT (T1)
      X2 = X - T23 * X1
      T1 = A1 * X2 + A2 * X1
      T2 = AINT (R23 * T1)
      Z = T1 - T23 * T2
      T3 = T23 * Z + A2 * X2
      T4 = AINT (R46 * T3)
      X = T3 - T46 * T4
      Y = R46 * X
      CALL timer(tend,tstart)
      myComp = myComp + tend
      RETURN
      END
C=====================================================================
```

A.2 Hough Transformation

The Hough transformation takes an input binary image of edge pixels and produces an array of edge parameters, indexed by its angle and radial distance from the origin.

```
      PROGRAM HoughProg
      USE HPF_LIBRARY
      IMPLICIT NONE
      INTEGER, PARAMETER :: NN = 512
      INTEGER, PARAMETER :: STDOUT = 6
      INTEGER, PARAMETER :: FILEIN = 20, FILEOUT=21
      REAL, PARAMETER :: PId2 = 1.5707963
C .. Define interface for subroutines ..
      INTERFACE
        SUBROUTINE timer(return_time, initial_time)
          REAL, INTENT(IN)  :: initial_time
          REAL, INTENT(OUT) :: return_time
        END SUBROUTINE timer
      END INTERFACE
C .. Arrays and scalars ..
      INTEGER, DIMENSION(NN,NN) :: Hough
      INTEGER, DIMENSION(:,:), ALLOCATABLE :: Edge
      INTEGER, DIMENSION(:,:), ALLOCATABLE :: XX,YY
      INTEGER, DIMENSION(:,:), ALLOCATABLE :: iRho
      INTEGER, DIMENSION(:,:), ALLOCATABLE :: iRhoInd2
      INTEGER, DIMENSION(:), ALLOCATABLE :: XXpacked, YYpacked
      INTEGER, DIMENSION(:,:), ALLOCATABLE :: CountBase
      LOGICAL, DIMENSION(:,:), ALLOCATABLE :: CountMask
      INTEGER :: i,j,k
      INTEGER :: EsizeX, EsizeY
      INTEGER :: NedgePixels
      INTEGER :: iargc
      REAL, DIMENSION(NN) :: COStheta, SINtheta
      REAL :: dTheta, dRho
      REAL :: ti,tf
      REAL :: tstart,tend
      REAL :: tcomms, tcomp
      CHARACTER*(32) magic
      CHARACTER*(80) filename
      LOGICAL, DIMENSION(:,:), ALLOCATABLE :: EdgeFilter
      LOGICAL doWrite
C .. HPF dirctives to distribute the arrays ..
!HPF$ PROCESSORS             :: procs(NUMBER_OF_PROCESSORS())
!HPF$ TEMPLATE,DIMENSION(NN) :: tplate
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: tplate
!HPF$ ALIGN (:) WITH tplate(:) :: COStheta,SINtheta
!HPF$ ALIGN (*,:) WITH tplate(:) :: Hough,CountBase
!HPF$ ALIGN (*,:) WITH tplate(:) :: iRho,iRhoInd2
!HPF$ ALIGN (*,:) WITH tplate(:) :: countMask
C Begin Executable
      tcomp = 0.0
      tcomms = 0.0
C Default values.
      EsizeX = 204
      EsizeY = 124
      doWrite = .TRUE.
C Read in image filename.
      filename = 'image.pbm'
C Read in the Edge image.
```

```
      OPEN(UNIT=FILEIN,FILE=filename,STATUS='UNKNOWN')
      READ(UNIT=FILEIN,FMT='(A2)') magic
      READ(UNIT=FILEIN,FMT=*)
      READ(UNIT=FILEIN,FMT=*) EsizeX, EsizeY
      ALLOCATE( Edge(EsizeX,EsizeY) )
      ALLOCATE( XX(EsizeX,EsizeY) )
      ALLOCATE( YY(EsizeX,EsizeY) )
      ALLOCATE( EdgeFilter(EsizeX,EsizeY) )
      WRITE(UNIT=STDOUT,FMT='(''Reading image '',I4,'' x '',I4)')
     1     EsizeX,EsizeY
      READ(UNIT=FILEIN,FMT=*) ( (Edge(i,j), i=1,EsizeX),j=1,EsizeY )
      CLOSE(UNIT=FILEIN)
      WRITE(UNIT=STDOUT,FMT='(''Starting transformation'')')
      NedgePixels = SUM( Edge )
      ALLOCATE( XXpacked(NedgePixels) )
      ALLOCATE( YYpacked(NedgePixels) )
      ALLOCATE( iRho(NedgePixels,NN) )
      ALLOCATE( iRhoInd2(NedgePixels,NN) )
C Start timer
      CALL timer(ti, 0.0)
      Hough = 0
C Set up XX() and YY() arrays, equal to their index numbers.
C Then mask out elements where the edge pixels equal zero.
      EdgeFilter = .FALSE.
      WHERE( Edge .EQ. 1 ) EdgeFilter = .TRUE.
      FORALL( i=1:EsizeX,j=1:EsizeY ) XX(i,j) = i - 1
      XXpacked = PACK( ARRAY=XX,MASK=EdgeFilter )
      FORALL( i=1:EsizeX,j=1:EsizeY ) YY(i,j) = j - 1
      YYpacked = PACK( ARRAY=YY,MASK=EdgeFilter )
      DEALLOCATE( EdgeFilter )
      DEALLOCATE( YY )
      DEALLOCATE( XX )
      DEALLOCATE( Edge )
C Set up COStheta and SINtheta coefficient arrays.
      dTheta = PId2 / (NN - 1)
      dRho   = (NN - 1) /
     1         SQRT( REAL(EsizeX*EsizeX) + REAL(EsizeY*EsizeY) )
      FORALL(k=1:NN) COStheta(k) = COS( (k-1)*dTheta )
      FORALL(k=1:NN) SINtheta(k) = SIN( (k-1)*dTheta )
C Set up iRhoInd2 array, to be used in COUNT_SCATTER() operation.
      FORALL( k=1:NN ) iRhoInd2(:,k) = k
C Parallel loop over 'theta' direction.
      FORALL( k=1:NN )
         iRho(:,k) = dRho*( XXpacked(:)*COStheta(k) +
     1                      YYpacked(:)*SINtheta(k) ) + 1.5
      END FORALL
C Set up tempoarary arrays for COUNT operation.
      ALLOCATE( CountMask( NedgePixels,NN ) )
      ALLOCATE( CountBase( NN,NN ) )
      CountMask = .TRUE.
      CountBase = 0
C Count up the occurences of iRho into Hough array.
C PGHPF on Cray requires 'ARRAY=' *not* to be present.  04/Feb/97.
      Hough = COUNT_SCATTER( CountMask, BASE=CountBase,
     1                       INDX1=iRho, INDX2=iRhoInd2 )
C Stop timer
      CALL timer(tf, ti)
C Adjust the timer.  It would appear from the profiler that no time
C was spent on communications...
      tcomms = 0.0
```

```
      tcomp  = tf
      IF( doWrite )THEN
         WRITE(UNIT=STDOUT,FMT='(''Writing Transform image'')')
         OPEN(UNIT=FILEOUT,FILE='hough.pgm',STATUS='UNKNOWN')
         WRITE(UNIT=FILEOUT,FMT='(A)') 'P2'
         WRITE(UNIT=FILEOUT,FMT='(A)')
     1      '# Created by Hough transformation program.  NPAC.'
         WRITE(UNIT=FILEOUT,FMT='(2I6)') NN,NN
         WRITE(UNIT=FILEOUT,FMT='(I6)') MAXVAL( Hough )
         WRITE(UNIT=FILEOUT,FMT=*) ((Hough(i,j), i=1,NN),j=1,NN)
         CLOSE(UNIT=FILEOUT)
      END IF
      WRITE(UNIT=*,FMT=6004) NUMBER_OF_PROCESSORS(),NN,
     1                       tcomms,tcomp,(tf-tcomms-tcomp),tf
      DEALLOCATE( CountBase )
      DEALLOCATE( CountMask )
      DEALLOCATE( iRhoInd2 )
      DEALLOCATE( iRho )
      DEALLOCATE( YYpacked )
      DEALLOCATE( XXpacked )
      STOP
 1001 FORMAT('Status: iTime = ',I6,' Error = ',F9.4)
 1002 FORMAT(12F9.3)
 6004 FORMAT('Number of Processors = ',I4/
     1       'Problem size = ',I5/
     2       'Communications = ',F9.3/
     3       'Compute        = ',F9.3/
     4       'Others         = ',F9.3/
     5       'Total time     = ',F9.3)
      END
C=====================================================================
```

A.3 Binomial Tree Stock Market Simulation

This is a binomial approximation stock market simulation model, incorporating stochastic volatility with American call (exercise at any time in contract) and with European call (exercise only at option maturity).

```
      PROGRAM main
      IMPLICIT NONE
C Define interfaces.
      INTERFACE
         SUBROUTINE timer(return_time, initial_time)
         REAL, INTENT(IN)  :: initial_time
         REAL, INTENT(OUT) :: return_time
         END SUBROUTINE timer
         SUBROUTINE asvcorp2(sp,sk,r,t,v,psi,xmu,cor,reps1,reps,div,
     1                       cval,tcomp,tcomms)
         REAL :: sp,sk,r,t,v,psi,xmu,cor,div,cval
         REAL :: tcomp, tcomms
         INTEGER reps,reps1
         END SUBROUTINE asvcorp2
         SUBROUTINE svcorp2(sp,sk,r,t,v,psi,xmu,cor,reps,cval,
     1                      tcomp,tcomms)
         REAL :: sp,sk,r,t,v,psi,xmu,cor,cval
         REAL :: tcomp, tcomms
         INTEGER reps
```

```
         END SUBROUTINE svcorp2
      END INTERFACE
      real sz,k,r,t,tdiv,div,v(4),rate
      real psi,xmu,rho,cbscall,camc,ceur,cams
      integer repb,reps,i, j, times, reps1
      REAL :: timediff,rtimes
      REAL :: ti,tf
      REAL :: tstart,tend
      REAL :: tcomms,tcomp
C Begin Executable
C Start timer
      CALL timer(ti, 0.0)
      tcomp  = 0.0
      tcomms = 0.0
      k = 100.00
      sz = 110
      repb = 100
      reps = 17
      psi = 1.0
      xmu = 0.0
      rate = 0.15
      rho = 0.234
      k = 100.00
      r = 4.50
      t = .25
      v(1) = 0.15
      v(2) = 0.15
      v(3) = 0.15
      v(4) = 0.15
      div = 1.0
      tdiv = 0.125
      times = 1
      rtimes = float(times)
c --- stochastic volatility European Model
      print *,'stochastic volatility European Model'
      print *,'--------------------------------------'
      print *,'reps           Value       Time(sec.)'
      do 1111 reps = 17, 17
         CALL timer(tstart, 0.0)
         do 6000 i = 1, times
            call svcorp2(sz,k,rate,t,v(3),psi,xmu,rho,reps,ceur,
     1                   tcomp,tcomms)
 6000    continue
         CALL timer(tend,tstart)
         timediff = tend / rtimes
         write(*,'(I3,9X,f9.4,5X,f8.4)') reps,ceur,timediff
 1111 continue
 9991 print *
C --- stochastic volatility American Model
      print *,'stochastic volatility American Model'
      print *,'reps   reps1       Value       Time(sec.)'
      print *,'----------------------------------------'
      do 2222 reps = 17, 17
         do 3333 j = 0, reps-1
            tdiv = t*j/float(reps)
            reps1=(INT(reps*tdiv/t)+1)
            CALL timer(tstart, 0.0)
            do 7000 i = 1, times
               call asvcorp2(sz,k,rate,t,v(4),psi,xmu,rho,reps1,reps,
     &              tdiv,cams,
```

```
     &              tcomp,tcomms)
 7000       continue
            CALL timer(tend,tstart)
            timediff = tend / rtimes
            write(*,'(I3,5X,I3,5X,f9.4,5X,f8.4)')
     &           reps,(INT(reps*tdiv/t)+1),
     &           cams,timediff
 3333    continue
 2222 continue
C Stop timer
      CALL timer(tf, ti)
      WRITE(UNIT=*,FMT=6004) NUMBER_OF_PROCESSORS(),reps,
     1                       tcomms,tcomp,(tf-tcomms-tcomp),tf
      STOP
 6004 FORMAT('Number of Processors = ',I4/
     1       'Problem size = ',I5/
     2       'Communications = ',F9.3/
     3       'Compute        = ',F9.3/
     4       'Others         = ',F9.3/
     5       'Total time     = ',F9.3)
      END
C=====================================================================
      SUBROUTINE asvcorp2(sp,sk,r,t,v,psi,xmu,cor,reps1,reps,div,cval,
     1                    tcomp,tcomms)
      IMPLICIT NONE
      REAL sp,sk,r,t,v,psi,xmu,cor,div,cval
      REAL :: tcomp, tcomms
      INTEGER reps,reps1
      REAL,DIMENSION(1:(2**(reps-12)-1),1:8192) :: ss1,vv1
      REAL,DIMENSION(1:2**(reps-13),1:8192) :: cearly_big
      REAL,DIMENSION(1:8192) :: aatmp1,aatmp2,uux,ddx
      REAL,DIMENSION(1:2**reps1) :: cearly,cback
      REAL u,d,dt,r1,tmp1,tmp2,rx,xreps
      INTEGER i,j,n,nreps1,start,end,repsdiv,reps2,nreps2
      REAL :: tstart,tend
!HPF$ PROCESSORS P( NUMBER_OF_PROCESSORS() )
!HPF$ DISTRIBUTE (*,BLOCK) ONTO P :: vv1,ss1,cearly_big
!HPF$ DISTRIBUTE (BLOCK) ONTO P:: uux,ddx,cearly,cback,aatmp1,aatmp2
      xreps = float(reps)
      dt = t/xreps
c       --- SET RX (PERIODIC 1+RF) TO RISK FREE RATE + 1
      rx=(r+1.0)**dt
      tmp1 = (xmu-0.5*psi*psi)*dt
      tmp2 = cor*psi*sqrt(dt)
      u = exp(tmp1+tmp2)
      d = exp(tmp1-tmp2)
c       --- create stock price lattice (sl)
      vv1(1,:) = 0.0
      ss1(1,:) = 0.0
      vv1(1,1) = v*v
      ss1(1,1) = sp
      n = 1
      do i = 1,13
         CALL timer(tstart, 0.0)
         aatmp1 = (r-0.5*vv1(1,:))*dt
         aatmp2 = sqrt(vv1(1,:)*dt)
         uux = exp(aatmp1+aatmp2)
         ddx = exp(aatmp1-aatmp2)
         CALL timer(tend, tstart)
         tcomp = tcomp + tend
```

```
    CALL timer(tstart, 0.0)
    vv1(1,:) =  u*vv1(1,:) +
&        eoshift(d*vv1(1,:),dim = 1,shift = -n)
    ss1(1,:) = uux*ss1(1,:) +
&        eoshift(ddx*ss1(1,:),dim = 1,shift = -n)
    CALL timer(tend, tstart)
    tcomms = tcomms + tend
    if (i.eq.reps1) then
       CALL timer(tstart, 0.0)
       cearly = ss1(1,1:2**i) - sk
       CALL timer(tend, tstart)
       tcomms = tcomms + tend
       CALL timer(tstart, 0.0)
       ss1(1,1:2**i) = ss1(1,1:2**i) - div
       CALL timer(tend, tstart)
       tcomp = tcomp + tend
    endif
    n = 2*n
 enddo
 if (reps.gt.13) then
    reps2 = reps - 13
    nreps1 = 2**(reps2+1) - 1
    repsdiv = reps1 - 14
    start = 2**repsdiv
    end = 2*start - 1
    CALL timer(tstart, 0.0)
    do i = 1, nreps1/2
       aatmp1 = (r-0.5*vv1(i,:))*dt
       aatmp2 = sqrt(vv1(i,:)*dt)
       uux = exp(aatmp1+aatmp2)
       ddx = exp(aatmp1-aatmp2)
       vv1(2*i,:) =  u*vv1(i,:)
       vv1((2*i+1),:) = d*vv1(i,:)
       if (i.ge.start.and.i.le.end) then
          aatmp1 = uux*ss1(i,:)
          aatmp2 = ddx*ss1(i,:)
          cearly_big(2*(i-start)+1,:) = aatmp1 - sk
          cearly_big(2*(i-start+1),:) = aatmp2 - sk
          ss1(2*i,:) =  aatmp1 - div
          ss1((2*i+1),:) = aatmp2 - div
       else
          ss1(2*i,:) = uux*ss1(i,:)
          ss1((2*i+1),:) = ddx*ss1(i,:)
       endif
       ss1(i,:) = 0.0
    enddo
    ss1 = ss1 - sk
    where (ss1.lt.0.0) ss1 = 0.0
    CALL timer(tend, tstart)
    tcomp = tcomp + tend
    if (reps1.le.13) then
       CALL timer(tstart, 0.0)
       do i=nreps1/2+1,nreps1
          ss1(1,:) = ss1(1,:) + ss1(i,:)
       enddo
       ss1(1,:) = ss1(1,:)/((2*rx)**(float(reps-13)))
       n = 1
       CALL timer(tend, tstart)
       tcomp = tcomp + tend
       CALL timer(tstart, 0.0)
```

```
            do i=1,13-reps1
               ss1(1,:) = ss1(1,:) +
     &              eoshift(ss1(1,:),dim = 1,shift = n)
               n = 2*n
            enddo
            CALL timer(tend, tstart)
            tcomms = tcomms + tend
            CALL timer(tstart, 0.0)
            nreps2 = 2**(13-reps1)
            cback = ss1(1,1:8192:nreps2)/
     &           ((2*rx)**float(13-reps1))
            where (cearly.gt.cback) cback = cearly
            CALL timer(tend, tstart)
            tcomp = tcomp + tend
            CALL timer(tstart, 0.0)
c       --- calculate call price
            cval = SUM(cback,mask=cback.gt.0.0)/((2*rx)**(float(reps1)))
            CALL timer(tend, tstart)
            tcomms = tcomms + tend
          else
            nreps2 = 2**(reps-reps1)
            cval = 0.0
            do i=1,2*start
               CALL timer(tstart, 0.0)
               aatmp1 = 0.0
               n = nreps1/2 + (i-1)*nreps2
               do j=1,nreps2
                  aatmp1 = aatmp1 +
     &                 ss1(n+j,:)
               enddo
               aatmp1 = aatmp1/((2*rx)**(float(reps-reps1)))
               where (cearly_big(i,:).gt.aatmp1)
     &              aatmp1 = cearly_big(i,:)
               CALL timer(tend, tstart)
               tcomp = tcomp + tend
               CALL timer(tstart, 0.0)
               cval = cval + SUM(aatmp1,mask=aatmp1.gt.0.0)
               CALL timer(tend, tstart)
               tcomms = tcomms + tend
            enddo
c       --- calculate call price
            cval = cval/((2*rx)**(float(reps1)))
          endif
       else
c       --- calculate call price
          ss1 = ss1 - sk
          CALL timer(tstart, 0.0)
          cval =  SUM(ss1(1,:),mask=ss1(1,:).gt.0.0)/((2*rx)**xreps)
          CALL timer(tend, tstart)
          tcomms = tcomms + tend
       endif
       return
       end
C=====================================================================
       SUBROUTINE svcorp2(sp,sk,r,t,v,psi,xmu,cor,reps,cval,
      1                   tcomp,tcomms)
       IMPLICIT NONE
       real sp,sk,r,t,v,psi,xmu,cor,cval
       REAL :: tcomp,tcomms
       integer reps
```

```
      real,dimension(1:(2**(reps-12)-1),1:8192) :: ss1,vv1,uux,ddx,
     &     aatmp1,aatmp2
      integer,dimension(1:8192) :: index,ud
      real u,d,dt,r1,tmp1,tmp2,rx,xreps
      integer i,reps1,nreps1,n2
      REAL :: tstart,tend
!HPF$ PROCESSORS P( NUMBER_OF_PROCESSORS() )
!HPF$ DISTRIBUTE (*,block) onto P:: ss1,vv1,uux,ddx,aatmp1,aatmp2
!HPF$ DISTRIBUTE (block) onto P:: index,ud
      CALL timer(tstart, 0.0)
      xreps = float(reps)
      dt = t/xreps
C..HWY.  Corrected syntax from 'index = [0:891]'.
      FORALL(i=1:8192) index(i) = i - 1
c       --- SET RX (PERIODIC 1+RF) TO RISK FREE RATE + 1
      rx=(r+1.0)**dt
      tmp1 = (xmu-0.5*psi*psi)*dt
      tmp2 = cor*psi*sqrt(dt)
      u = exp(tmp1+tmp2)
      d = exp(tmp1-tmp2)
c       --- create stock price lattice (sl)
      vv1(1,:) = v*v
      ss1(1,:) = sp
      n2 = 4096
      do i = 1,13
         aatmp1(1,:) = (r-0.5*vv1(1,:))*dt
         aatmp2(1,:) = sqrt(vv1(1,:)*dt)
         uux(1,:) = exp(aatmp1(1,:)+aatmp2(1,:))
         ddx(1,:) = exp(aatmp1(1,:)-aatmp2(1,:))
         ud = index/n2
c         where (mod(ud,2) .ne. 0)
         where (BTEST(ud,POS=0))
            vv1(1,:) =  d*vv1(1,:)
            ss1(1,:) = ddx(1,:)*ss1(1,:)
         elsewhere
            vv1(1,:) =  u*vv1(1,:)
            ss1(1,:) = uux(1,:)*ss1(1,:)
         endwhere
         n2 = n2/2
      enddo
      CALL timer(tend, tstart)
      tcomp = tcomp + tend
      if (reps.gt.13) then
         CALL timer(tstart, 0.0)
         reps1 = reps - 13
         nreps1 = 2**(reps1+1) - 1
         do i = 1, nreps1/2
            aatmp1(i,:) = (r-0.5*vv1(i,:))*dt
            aatmp2(i,:) = sqrt(vv1(i,:)*dt)
            uux(i,:) = exp(aatmp1(i,:)+aatmp2(i,:))
            ddx(i,:) = exp(aatmp1(i,:)-aatmp2(i,:))
            vv1(2*i,:) =  u*vv1(i,:)
            vv1((2*i+1),:) = d*vv1(i,:)
            ss1(2*i,:) = uux(i,:)*ss1(i,:)
            ss1((2*i+1),:) = ddx(i,:)*ss1(i,:)
            ss1(i,:) = 0.0
         enddo
c       --- calculate call price
         ss1 = ss1 - sk
         CALL timer(tend, tstart)
```

```
         tcomp = tcomp + tend
         CALL timer(tstart, 0.0)
         cval =  SUM(ss1,mask=ss1.gt.0.0)/((2*rx)**xreps)
         CALL timer(tend, tstart)
         tcomms = tcomms + tend
       else
c        --- calculate call price
         ss1 = ss1 - sk
         CALL timer(tstart, 0.0)
         cval =  SUM(ss1(1,:),mask=ss1(1,:).gt.0.0)/((2*rx)**xreps)
         CALL timer(tend, tstart)
         tcomms = tcomms + tend
       endif
       return
       end
```

A.4 Gaussian Elimination

This code performs factorization of a random matrix, through Gaussian elimination with partial pivoting.

```
      PROGRAM gauss
      IMPLICIT NONE
      INTEGER, PARAMETER :: Nrows = 255, Ncols = 256
C Define interface for subroutine timer.
      INTERFACE
        SUBROUTINE timer(return_time, initial_time)
          REAL, INTENT(IN)  :: initial_time
          REAL, INTENT(OUT) :: return_time
        END SUBROUTINE timer
      END INTERFACE
C Mask, for setting which rows in which to search for next pivot.
      INTEGER, DIMENSION(Nrows) :: pivotMask
      DOUBLE PRECISION, DIMENSION(Nrows,Ncols) :: a,pivotColArray
      DOUBLE PRECISION, DIMENSION(Nrows) :: tmpCol,x
      DOUBLE PRECISION, DIMENSION(Ncols) :: pivotRow
C Matrix row, column, and pivot-stage indices.
      INTEGER :: i,j,k
      INTEGER :: ipivot
      INTEGER, DIMENSION(1) :: ipivotArray
      DOUBLE PRECISION :: pivot,tmp
      REAL :: ti,tf
      REAL :: tstart,tend
      REAL :: tcomms, tcomp
C HPF Directives
!HPF$ PROCESSORS procs(NUMBER_OF_PROCESSORS())
!HPF$ TEMPLATE temp(Ncols)
!HPF$ DISTRIBUTE temp(CYCLIC)
!HPF$ ALIGN a(*,j)             WITH temp(j)
!HPF$ ALIGN pivotColArray(*,j) WITH temp(j)
!HPF$ ALIGN pivotRow(j)        WITH temp(j)
C Executable Code.
      tcomp = 0.0
      tcomms = 0.0
C Generate random matrix.
      CALL RANDOM_NUMBER( a )
C Calculate appropriate RHS into Ncols-th column of a(), such
```

```
C that the solution should equal to 1.0,2.0,3.0,...Nrows.
      FORALL( i=1:Nrows ) x(i) = REAL(i)
      tmpCol = 0.0D0
      DO i=1,Nrows
        FORALL( j=1:Nrows ) tmpCol(j) = a(i,j)*x(j)
        a(i,Ncols) = SUM( tmpCol )
      END DO
      x = 0.0D0
C Start timer
      CALL timer(ti, 0.0)
      pivotMask = 1
      DO k=1,Nrows-1
C Use tmpCol as temporary column to find the pivot.
        CALL timer(tstart,0.0)
        tmpCol = ABS( a(:,k) )
        CALL timer(tend,tstart)
        tcomms = tcomms + tend
C Find the pivot in the k-th column, within the rows k:Nrows.
        ipivotArray = MAXLOC( tmpCol,MASK=pivotMask.EQ.1 )
        ipivot = ipivotArray(1)
        IF( ipivot .NE. k )THEN
C Swap rows, using pivotRow as a temporary storage.
          pivotRow(:) = a(ipivot,:)
          a(ipivot,:) = a(k,:)
          a(k,:) = pivotRow(:)
        ELSE
          pivotRow(:) = a(k,:)
        END IF
C Define the pivot, and the (rescaled) pivotColArray.  This is the
C pivot column replicated across all columns.
        CALL timer(tstart,0.0)
        pivot = a(k,k)
        FORALL( i=k+1:Nrows,j=1:Ncols )
     $         pivotColArray(i,j) = a(i,k)/pivot
        CALL timer(tend,tstart)
        tcomms = tcomms + tend
C Calculate upper triangular factor, by updating matrix using the
C pivot, pivotRow and pivotColarray.
        FORALL( i=k+1:Nrows,j=k+1:Ncols )
     $          a(i,j) = a(i,j) - pivotColArray(i,j)*pivotRow(j)
C Save pivot column for the lower triangular factor.
        a(k+1:Nrows,k) = pivotColArray(k+1:Nrows,k)
        pivotMask(k) = 0
      END DO
C Now do backwards substitution.
      DO i=Nrows,1,-1
        CALL timer(tstart,0.0)
        tmp = a(i,Ncols)
        CALL timer(tend,tstart)
        tcomms = tcomms + tend
        tmpCol = 0.0D0
        CALL timer(tstart,0.0)
        FORALL( j=1:Nrows,j.GT.i ) tmpCol(j) = a(i,j)*x(j)
        CALL timer(tend,tstart)
        tcomms = tcomms + tend
        tmp = tmp - SUM( tmpCol )
        CALL timer(tstart,0.0)
        x(i) = tmp/a(i,i)
        CALL timer(tend,tstart)
        tcomms = tcomms + tend
```

```
      END DO
C Stop timer
      CALL timer(tf, ti)
      tcomp = tf - ti - tcomms
      WRITE(UNIT=*,FMT=*) 'Solution is:'
      WRITE(UNIT=*,FMT=101) (x(i),i=1,Nrows)
      WRITE(UNIT=*,FMT=6004) NUMBER_OF_PROCESSORS(),Ncols,
     1                       tcomms,tcomp,(tf-tcomms-tcomp),tf
      STOP
 101  FORMAT(10F9.3)
 6004 FORMAT('Number of Processors = ',I4/
     1       'Problem size = ',I5/
     2       'Communications = ',F9.3/
     3       'Compute        = ',F9.3/
     4       'Others         = ',F9.3/
     5       'Total time     = ',F9.3)
      END
C=====================================================================
```

A.5 Sweep Over an Unstructured Mesh

This code represents a simple edge sweep on an unstructured mesh. The additional flux at node 'i' is the sum of all its neighbouring nodes. The edge array '`ndeE()`' stores the adjacency list. The amount of communication generated will depend on the locality of access. The cost of the communication will be low if the nodes are ordered such that most of these accesses are local. This reordering can be potentially achieved by use of extrinsic functions to perform graph partitioning using the edge array.

```
      PROGRAM sweep
      USE HPF_LIBRARY
C     Sweep over an unstructured mesh
      INTEGER, PARAMETER :: ne=3000,nn=500,iter=100
      INTEGER ndeS(ne),ndeE(ne)
      INTEGER i,j
      REAL    flux(nn),econs
      REAL    weight(nn,4),cc(nn)
      REAL    data(ne)
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE ndeS(BLOCK) ONTO P
!HPF$ ALIGN ndeE(:)  WITH  ndeS(:)
!HPF$ ALIGN flux(:)  WITH  weight(:,*)
!HPF$ ALIGN cc(:)    WITH  weight(:,*)
!HPF$ DISTRIBUTE weight(BLOCK,*) ONTO P
C     Initialization
      FORALL(i=1:nn) flux(i) = 0.0
      FORALL(i=1:nn) weight(i,1) = 1.0
      FORALL(i=1:nn) weight(i,2) = 0.5
      FORALL(i=1:nn) weight(i,3) = 0.5
      FORALL(i=1:nn) weight(i,4) = 1.0
      FORALL(i=1:nn) cc(i) = 1.0
      econs = 1.0
      FORALL(i=1:ne)
         ndeS(i) = 1+MOD((i-1),nn)
         ndeE(i) = 1+MOD(ne-(i-1),nn)
      END FORALL
```

```
      DO j = 1,iter
C     Weighted Sweep over edges
         FORALL(i=1:ne)
            data(i) = econs*(
     &          ((weight(ndeS(i),2)+weight(ndeS(i),3)+weight(ndeS(i),4))
     &          / weight(ndeS(i),1)
     &          +
     &           (weight(ndeE(i),2)+weight(ndeE(i),3)+weight(ndeE(i),4))
     &          / weight(ndeE(i),1)
     &          ) / 2 + cc(ndeS(i)) )
         END FORALL
         flux = SUM_SCATTER(data,flux,ndeS)
         FORALL(i=1:ne)
            data(i) = econs*(
     &          ((weight(ndeS(i),2)+weight(ndeS(i),3)+weight(ndeS(i),4))
     &          / weight(ndeS(i),1)
     &          +
     &           (weight(ndeE(i),2)+weight(ndeE(i),3)+weight(ndeE(i),4))
     &          / weight(ndeE(i),1)
     &          ) / 2 + cc(ndeE(i)) )
         END FORALL
         flux = SUM_SCATTER(data,flux,ndeE)
      END DO
      DO i=1, nn
         PRINT *,'flux(',i,') = ',flux(i)
      END DO
      END
```

A.6 2-Dimensional Potts Model

This code sets up a 2-D Q-State Potts model, and evolves it using the classic Metropolis heat-bath algorithm.

```
      PROGRAM potts2d
      IMPLICIT NONE
      integer, parameter :: m = 64 ! Lattice dimensions
      integer, parameter :: n = 64
      real ti,start, tf,tend,tcom
      real correct1,correct2
      real tcalc
!HPF$ PROCESSORS PROCS(NUMBER_OF_PROCESSORS())
!HPF$ TEMPLATE, DIMENSION(m,n) :: rectangle
!HPF$ DISTRIBUTE (*,BLOCK) ONTO PROCS :: rectangle
      interface
         subroutine timer(return_time, initial_time)
         real, intent(in)  :: initial_time
         real, intent(out) :: return_time
         end subroutine timer
         subroutine setup_random(iseed)
         integer, intent(IN) :: iseed
         end subroutine setup_random
      end interface
C     Number of allowable States for Degrees of Freedom
      integer, parameter :: q = 4
C     Coupling in kT units, -ve for ferromagnetic
      real, parameter :: cup = -4.0
C     The lattice of DOF's
```

```
      integer, dimension( 1:m, 1:n ) :: latt
!HPF$ ALIGN (:,:) WITH RECTANGLE :: latt
C     Trial choices for DOF's
      integer, dimension( 1:m, 1:n ) :: itmp
!HPF$ ALIGN (:,:) WITH RECTANGLE :: itmp
C     TMP for RNG and Probability
      real, dimension( 1:m, 1:n ) :: tmp, p
!HPF$ ALIGN (:,:) WITH RECTANGLE :: tmp
!HPF$ ALIGN (:,:) WITH RECTANGLE :: p
C     mask used in ensuring detailed balance prevails
      logical, dimension( 1:m, 1:n ) :: mask
C     !HPF$ ALIGN mask WITH RECTANGLE
C     number of measurements
      integer :: nmeasure = 20
C     number of sweeps per measurement
      integer :: niter = 10
      integer  imeasure,iter
      integer  i0,i,j,j0
C Begin Executable
      tcom = 0.0
      tcalc = 0.0
      latt = 0
      itmp = 0
      tmp = 0.0
      p = 0.0
      mask = .FALSE.
C     BEGIN TIMER
      call timer(start,0.0)
      call timer(ti,0.0)
      call setup_random( 12345 )
C     initialise between 0 and q-1 inclusive
      call random_number( tmp )
      latt = int( tmp * q )
      call timer(tcalc,ti)
      do imeasure=1,nmeasure
C     Measurements:
         call timer(ti,0.0)
C     Present bonds
         itmp =        merge( 1, 0, latt == cshift(latt,  1, 1 ) )
         itmp = itmp + merge( 1, 0, latt == cshift(latt, -1, 1 ) )
         call timer(tend,ti)
         tcalc=tcalc + tend
         call timer(ti,0.0)
         itmp = itmp + merge( 1, 0, latt == cshift(latt,  1, 2 ) )
         itmp = itmp + merge( 1, 0, latt == cshift(latt, -1, 2 ) )
         call timer(tend,ti)
         tcom= tcom + tend
         write(6,*) '# Bonds = ', sum( itmp )
         do iter=1,niter
            call timer(ti,0.0)
C     generate "new" values for the Potts variables
            call random_number(tmp)
C     using the Coddington recipe
            itmp = mod( (latt + int( tmp * q-1 ) + 1 ), q )
            call timer(tend,ti)
            tcalc=tcalc+tend
C     use four-colour ordering to preserve detailed-balance
            do i0=1,2
               do j0=1,2
                  call timer(ti,0.0)
```

```
C     have to get too many cos of this stupid non-function RNG
                call random_number( tmp )
                mask = .false.
                forall(i=i0:m:2, j=j0:n:2 ) mask(i,j) = .true.
                where (mask)
C     Present bonds
                   p =     merge( cup, 0.0,
     1                            latt == cshift(latt,  1, 1 ) )
                   p = p + merge( cup, 0.0,
     1                            latt == cshift(latt, -1, 1 ) )
                end where
                call timer(tend,ti)
                tcalc=tcalc+tend
                call timer(ti,0.0)
                where (mask)
                   p = p + merge( cup, 0.0,
     1                            latt == cshift(latt,  1, 2 ) )
                   p = p + merge( cup, 0.0,
     1                            latt == cshift(latt, -1, 2 ) )
                end where
                call timer(tend,ti)
                tcom=tcom+tend
                call timer(ti,0.0)
                where (mask)
C     future bonds
                   p = p - merge( cup, 0.0,
     1                            itmp == cshift(latt,  1, 1 ) )
C     if dof is flipped
                   p = p - merge( cup, 0.0,
     1                            itmp == cshift(latt, -1, 1 ) )
                end where
                call timer(tend,ti)
                tcalc=tcalc+tend
                call timer(ti,0.0)
                where (mask)
                   p = p - merge( cup, 0.0,
     1                            itmp == cshift(latt,  1, 2 ) )
                   p = p - merge( cup, 0.0,
     1                            itmp == cshift(latt, -1, 2 ) )
                end where
                call timer(tend,ti)
                tcom=tcom+tend
                call timer(ti,0.0)
                where (mask)
                   p = exp( p )
                   latt = merge( itmp, latt, tmp < p )
                end where
                call timer(tend,ti)
                tcalc=tcalc+tend
             end do
          end do              ! end of checkerboard loop
       end do                 ! end of iter loop
      end do                  ! end of measure loop
      call timer(tf,start)
C     END TIMER
      write(6,6001) NUMBER_OF_PROCESSORS(),
     1     m,tcom,tcalc,(tf-tcom-tcalc),tf
 6001 FORMAT('Number of Processors =',I4/
     1     'Problem size =',I4/
     2     'Communications = ',F9.3/
```

```
     3     'Compute = ', F9.3/
     4     'Others =', F9.3/
     5     'Total time =',F9.3 )
C write out picture of configuration as a PPM file
      write(9,100)
 100  Format('P3' )
      write(9,101)
 101  Format('# 2d Potts Model')
      write(9,FMT='(2I6)') m, n
      write(9,FMT='(I6)') 255
      do i=1,m
         do j=1,n
            if( latt(i,j) == 0 )then
               write(9,'(3I4)',ADVANCE='NO')
     1                         255,0,0
            else if( latt(i,j) == 1 )then
               write(9,'(3I4)',ADVANCE='NO')
     1                         0,255,0
            else if( latt(i,j) == 2 )then
               write(9,'(3I4)',ADVANCE='NO')
     1                         0,0,255
            else if( latt(i,j) == 3 )then
               write(9,'(3I4)',ADVANCE='NO')
     1                         255,255,0
            else if( latt(i,j) == 4 )then
               write(9,'(3I4)',ADVANCE='NO')
     1                         255,0,255
            else if( latt(i,j) == 5 )then
               write(9,'(3I4)',ADVANCE='NO')
     1                         0,255,255
            else if( latt(i,j) == 6 )then
               write(9,'(3I4)',ADVANCE='NO')
     1                         255,255,255
            else
               write(9,'(3I4)',ADVANCE='NO')
     1                         0,0,0
            endif
C Insert line break, as required by Cray Fortran.
            if( MOD( ((i-1)*n)+j,5 ) .EQ. 0 ) write(9,FMT=*)
         enddo
      enddo
      end program
      subroutine setup_random(iseed)
C     put a new seed in the bloody awful system supplied RNG
      IMPLICIT NONE
      integer :: n,i,iseed
      integer, dimension(:), allocatable ::  seed
      call random_seed(size=n)
      allocate( seed(n) )
      forall(i=1:n) seed(i) = mod(i*4096+150889+iseed,714025 )
      call random_seed( put=seed(1:n) )
      deallocate( seed )
      end subroutine
      subroutine timer(return_time, initial_time)
      real, intent(in)  :: initial_time
      real,  intent(out) :: return_time
      integer finish,rate
      CALL system_clock (COUNT=finish,COUNT_RATE=rate)
      return_time = FLOAT(finish) / FLOAT(rate) - initial_time
      RETURN
```

```
      END
```

A.7 2-D Binary Phase Quenching of Cahn Hilliard Cook Equation

This code sets up a random binary phase mixture and applies the Cahn Hilliard Cook equation to quench the system into a cooled state.

```
      PROGRAM chc
      IMPLICIT NONE
C The mesh size
      integer, parameter :: n1 = 256
      integer, parameter :: n2 = 256
C number of bins to measure
      integer, parameter :: nb = 20
      real, dimension(1:nb) :: bins
      real, dimension(n1,n2) :: c, f
      real ti, tf,start,tcalc,tcom,a,others
      real cmax,less
!HPF$ PROCESSORS :: PROCS(NUMBER_OF_PROCESSORS())
!HPF$ TEMPLATE, DIMENSION(n2) :: TEMP
!HPF$ DISTRIBUTE (BLOCK) ONTO PROCS :: TEMP
!HPF$ ALIGN (*,:) WITH TEMP(:) :: c,f
      interface
         subroutine timer(return_time, initial_time)
         real, intent(in)  :: initial_time
         real, intent(out) :: return_time
         end subroutine
      end interface
      real t1, t2, x
      real :: dt, dtby2
      integer :: iter
      integer :: niter
      integer i,j
C Executable Code.
      c = 0.0
      f = 0.0
      bins = 0.0
      OPEN(UNIT=3,FILE='in.runme',STATUS='OLD')
      READ(UNIT=3,FMT=*) dt
      READ(UNIT=3,FMT=*) niter
      dtby2 = dt / 2.0
      WRITE(UNIT=6,FMT='(''DT = '',F9.3,'' niter = '',I12)') dt,niter
      tcalc=0.0
      tcom=0.0
      call timer(start,0.0)
      call timer(ti,0.0)
C Use the (poor) system supplied RNG
      c = 1.0
      call random_number( c )
      c = c - 0.5
      call timer(tcalc,ti)
      do iter=1,niter
        call timer(ti,0.0)
        f = -1.0*(cshift(c,1,1) + cshift(c,-1,1))
        f = 3.0 * c +  c**3 + f
        call timer(a,ti)
        tcalc=tcalc+a
```

```
        call timer(ti,0.0)
        f= f- (cshift(c,1,2) + cshift(c,-1,2))
        call timer(a,ti)
        tcom=tcom+a
        call timer(ti,0.0)
        c = c + dtby2 * ( cshift(f,1,1) + cshift(f,-1,1)-4.0*f)
        call timer(a,ti)
        tcalc=tcalc+a
        call timer(ti,0.0)
        c = c + dtby2*(cshift(f,1,2) + cshift(f,-1,2))
        call timer(a,ti)
        tcom=tcom+a
        call timer(ti,0.0)
C be cautious in case of leaks
        cmax = maxval( abs(c) )
        if( cmax > 0.0 ) c = c / cmax
        call timer(a,ti)
        tcalc=tcalc+a
      end do
      call timer(tf,start)
      others=tf-tcalc-tcom
      write(6,6001) NUMBER_OF_PROCESSORS(),
     1              n1,tcom,tcalc,others,tf
 6001 FORMAT('Number of Processors =',I4/
     1 'Problem size =',I4/
     2 'Communications = ',F9.3/
     3 'Compute = ', F9.3/
     4 'Others =', F9.3/
     5 'Total time =',F9.3 )
      STOP
      end program
C=====================================================================
      subroutine timer(return_time, initial_time)
      real,  intent(in)  :: initial_time
      real,  intent(out) :: return_time
      integer finish,rate
      CALL system_clock( COUNT=finish,COUNT_RATE=rate)
      return_time= FLOAT(finish) / FLOAT(rate) - initial_time
      RETURN
      end subroutine
C=====================================================================
```

A.8 Solution of 2-D Poisson Equation by the Alternating Direct Implicit Method

This code illustrates the use of HPF constructs for fluid dynamics problems. The main program relies on two subroutines, for solving a tridiagonal matrix given a right hand side, and for setting up the right hand side itself.

A.8.1 Main program for the ADI code

```
      PROGRAM adi
      IMPLICIT NONE
      INCLUDE 'adi.inc'
C .. Define interface for subroutines ..
```

```
      INTERFACE
        SUBROUTINE tridiag_psi(alpha,beta,rhs,u,temp,N)
          INTEGER,INTENT(IN)              :: N
          REAL,INTENT(IN)                 :: alpha
          REAL,DIMENSION(:,:),INTENT(IN)  :: rhs
          REAL,DIMENSION(:,:),INTENT(OUT) :: u,temp,beta
!HPF$          DISTRIBUTE *(*,BLOCK) :: beta,rhs,u,temp
        END SUBROUTINE tridiag_psi
        SUBROUTINE find_rhs_psi(alpha,h2,psi,zeta,rhs,N)
          INTEGER,INTENT(IN)              :: N
          REAL,INTENT(IN)                 :: alpha,h2
          REAL,DIMENSION(:,:),INTENT(IN)  :: psi,zeta
          REAL,DIMENSION(:,:),INTENT(OUT) :: rhs
!HPF$     DISTRIBUTE *(*,BLOCK) :: psi,zeta,rhs
        END SUBROUTINE find_rhs_psi
        SUBROUTINE timer(return_time, initial_time)
          REAL, INTENT(IN)  :: initial_time
          REAL, INTENT(OUT) :: return_time
        END SUBROUTINE timer
      END INTERFACE
C .. Arrays and scalars ..
      INTEGER               :: i,j
      INTEGER               :: iTime
      REAL :: ti,tf
      REAL :: tstart,tend
      REAL :: tcomms, tcomp
C The number of time steps to iterate for.
      INTEGER               :: NTime
      REAL                  :: alpha,h2
      REAL                  :: error
      REAL,DIMENSION(NN,NN) :: beta,psi,zeta,rhs,wspace
      REAL,DIMENSION(NN,NN) :: tarray
C .. HPF dirctives to distribute arrays ..
!HPF$ PROCESSORS             :: procs(NUMBER_OF_PROCESSORS())
!HPF$ TEMPLATE,DIMENSION(NN) :: tplate
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: tplate
!HPF$ ALIGN beta(*,:)   WITH tplate(:)
!HPF$ ALIGN psi(*,:)    WITH tplate(:)
!HPF$ ALIGN zeta(*,:)   WITH tplate(:)
!HPF$ ALIGN rhs(*,:)    WITH tplate(:)
!HPF$ ALIGN wspace(*,:) WITH tplate(:)
!HPF$ ALIGN tarray(*,:) WITH tplate(:)
C Begin Executable
C Start timer
      CALL timer(ti, 0.0)
      tcomp = 0.0
      tcomms = 0.0
C .. Initialise scalars ..
      h2    = H*H
      alpha = 2.0*h2/DT
      NTime = 20
C .. Initialise arrays ..
      rhs    = 1.0
      wspace = 0.0
      psi    = 5.0
      psi (2:NN-1,2:NN-1) = 2.0
C .. Reduce to Laplace equation.
      zeta = 0.0
C .. Solve, over a number of time steps ..
      DO iTime = 1,NTime
```

```
        CALL timer(tstart, 0.0)
        tarray = (psi - 5.0) * (psi - 5.0)
        CALL timer(tend, tstart)
        tcomp = tcomp + tend
        CALL timer(tstart, 0.0)
        error = SUM( tarray ) / (NN*NN)
        CALL timer(tend, tstart)
        tcomms = tcomms + tend
        WRITE(UNIT=STDOUT,FMT=1001) iTime, error
        IF( error .LE. eTol ) EXIT
C .. Solve in x-direction.
        CALL timer(tstart, 0.0)
        CALL tridiag_psi(alpha,beta,rhs,psi,wspace,NN)
        CALL find_rhs_psi(alpha,h2,psi,zeta,rhs,NN)
        CALL timer(tend, tstart)
        tcomp = tcomp + tend
C.. Transpose
        CALL timer(tstart, 0.0)
        psi    = TRANSPOSE(psi)
        zeta   = TRANSPOSE(zeta)
        rhs    = TRANSPOSE(rhs)
        CALL timer(tend, tstart)
        tcomms = tcomms + tend
C .. Solve in y-direction.
        CALL timer(tstart, 0.0)
        CALL tridiag_psi(alpha,beta,rhs,psi,wspace,NN)
        CALL find_rhs_psi(alpha,h2,psi,zeta,rhs,NN)
        CALL timer(tend, tstart)
        tcomp = tcomp + tend
C.. Transpose
        CALL timer(tstart, 0.0)
        psi    = TRANSPOSE(psi)
        zeta   = TRANSPOSE(zeta)
        rhs    = TRANSPOSE(rhs)
        CALL timer(tend, tstart)
        tcomms = tcomms + tend
      END DO ! iTime
C Stop timer
      CALL timer(tf, ti)
      WRITE(UNIT=STDOUT,FMT=*) 'Final Result: for psi()'
      tarray = (psi - 5.0) * (psi - 5.0)
      error = SUM( tarray ) / (NN*NN)
      WRITE(UNIT=STDOUT,FMT=1001) NTime+1, error
      WRITE(UNIT=*,FMT=6004) NUMBER_OF_PROCESSORS(),NN,
     1                       tcomms,tcomp,(tf-tcomms-tcomp),tf
      STOP
 1001 FORMAT('Status: iTime = ',I6,' Error = ',F9.4)
 1002 FORMAT(12F9.3)
 6004 FORMAT('Number of Processors = ',I4/
     1       'Problem size = ',I5/
     2       'Communications = ',F9.3/
     3       'Compute        = ',F9.3/
     4       'Others         = ',F9.3/
     5       'Total time     = ',F9.3)
      END
C======================================================================
```

A.8.2 Routine to solve a tridiagonal matrix, for the ADI code

```
      SUBROUTINE tridiag_psi(alpha,beta,rhs,u,temp,N)
      IMPLICIT NONE
C .. Arguments ..
      INTEGER,INTENT(IN)              :: N
      REAL,INTENT(IN)                 :: alpha
      REAL,DIMENSION(:,:),INTENT(IN)  :: rhs
      REAL,DIMENSION(:,:),INTENT(OUT) :: u,temp,beta
C .. HPF directives ..
!HPF$ DISTRIBUTE *(*,BLOCK) :: beta,rhs,u,temp
C .. Local variables ..
      INTEGER i,k,j1,jN
      REAL a,b,c
C Begin Executable
      j1 = 2                          ! Start element to update
      jN = N-1                        ! End index to update
C Define matrix elements.
      a = -1.0                        ! Lower diagonal elements.
      b = alpha + 2.0                 ! Diagonal elements.
      c = -1.0                        ! Upper diagonal elements.
C Do LU decomposition and forward substitution.
      u(j1,j1:jN)    = ( rhs(j1,j1:jN) - a*u(1,j1:jN) )/b
      beta(j1,j1:jN) = b
      DO k=j1+1,jN
        temp(k,j1:jN) = c/beta(k-1,j1:jN)
        beta(k,j1:jN) = b - a*temp(k,j1:jN)
        u(k,j1:jN)    = (rhs(k,j1:jN) - a*u(k-1,j1:jN))/beta(k,j1:jN)
      END DO
      temp(N,j1:jN) = c/beta(jN,j1:jN)
C Do backwards substitution.
      DO k=jN,j1,-1
        u(k,j1:jN) = u(k,j1:jN) - temp(k+1,j1:jN)*u(k+1,j1:jN)
      END DO
      RETURN
      END SUBROUTINE tridiag_psi
C======================================================================
```

A.8.3 Routine to find the Right Hand Side, for the ADI code

```
      SUBROUTINE find_rhs_psi(alpha,h2,psi,zeta,rhs,N)
      IMPLICIT NONE
C .. Arguments ..
      INTEGER,INTENT(IN)              :: N
      REAL,INTENT(IN)                 :: alpha,h2
      REAL,DIMENSION(:,:),INTENT(IN)  :: psi,zeta
      REAL,DIMENSION(:,:),INTENT(OUT) :: rhs
C .. HPF directives ..
!HPF$ DISTRIBUTE *(*,BLOCK) :: psi,zeta,rhs
C .. Local variables ..
      INTEGER i,k,j1,jN
      REAL b
C Begin Executable
      j1 = 2                          ! Start element to update
      jN = N-1                        ! End index to update
      b = alpha - 2.0                 ! Diagonal elements
      DO k=j1,jN
        rhs(k,j1:jN) = psi(k-1,j1:jN) + b*psi(k,j1:jN) + psi(k+1,j1:jN)
     $               - h2*zeta(k,j1:jN)
```

```
      END DO
C Set boundary conditions
      rhs(1,:) = psi(1,:)
      rhs(N,:) = psi(N,:)
      DO k=j1,jN
        rhs(k,1) = psi(k,1)
        rhs(k,N) = psi(k,N)
      END DO
      RETURN
      END SUBROUTINE find_rhs_psi
C======================================================================
```

A.9 Direct N-Body Solver

This code calculates the full-range interactions amongst a number of particles, with no cut-off. The code also avoids superfluous computations by exploiting Newton's third law to halve the possible computations.

```
      PROGRAM nbody
      IMPLICIT NONE
C ----------------------------------------------------------------------
C|                                                                      |
C| Direct N-body simulation code.                                       |
C| Computes the forces between N-bodies after one time-step.            |
C|                                                                      |
C| MM: mass.                                                            |
C| PX, PY, PZ: position in x, y, z coordinates.                         |
C| FX, FY, FZ: force in x, y, z coordinates.                            |
C|                                                                      |
C| H W Yau.  25th of August, 1996.                                      |
C| Northeast Parallel Architectures Center.                             |
C| Syracuse University.                                                 |
C|______________________________________________________________________|
C|                                                                      |
C| Edit record:                                                         |
C| 06/Jun/96 Cleaned up by M. McMahon.  Modified all the array          |
C|           statements to dimension.                                   |
C|           Modified by M.McMahon 6/6/96 Changed EXTRINSIC and ALIGN    |
C|           statements.                                                |
C| 17/Jun/96 (MM) fixed up timer.                                       |
C| 22/Aug/1996: (HWY) Made problem smaller from NN=4096.                |
C|              Fixed code to remove hard-wiring of '4096'.             |
C| 25/Aug/1996: Tidying up.                                             |
C|              Removed superfluous 'NP' parameter.                     |
C|              Fortran 90 version of the same code.                    |
C| 08/Apr/1997: (HWY) Bug fixes.                                        |
C|              Rewrote initialisation of states, to be predictable.    |
C| 09/Apr/1997: Use of allocatable arrays.                              |
C|______________________________________________________________________|
      INTERFACE
         SUBROUTINE timer(return_time, initial_time)
         REAL, INTENT(IN)  :: initial_time
         REAL, INTENT(OUT) :: return_time
         END SUBROUTINE
      END INTERFACE
      INTEGER, PARAMETER :: NN = 3200
      INTEGER, PARAMETER :: MM=0, PX=1, PY=2, PZ=3, FX=4, FY=5, FZ=6
```

```
      REAL, PARAMETER :: G = 1.0
      REAL,DIMENSION(:),ALLOCATABLE :: dx,dy,dz,tx,ty,tz,
     1                                 sq,dist,fac
      REAL,DIMENSION(:,:),ALLOCATABLE :: p,q
      INTEGER :: i,j,k,npts
      REAL    :: ti, tf,tcalc,tcom,start,a
C HPF directives
!HPF$ PROCESSORS :: procs(NUMBER_OF_PROCESSORS())
!HPF$ TEMPLATE,DIMENSION(0:NN-1) :: tplate
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: tplate
!HPF$ ALIGN dx(:)  WITH tplate(:)
!HPF$ ALIGN dy(:)  WITH tplate(:)
!HPF$ ALIGN dz(:)  WITH tplate(:)
!HPF$ ALIGN tx(:)  WITH tplate(:)
!HPF$ ALIGN ty(:)  WITH tplate(:)
!HPF$ ALIGN tz(:)  WITH tplate(:)
!HPF$ ALIGN sq(:)  WITH tplate(:)
!HPF$ ALIGN dist(:)WITH tplate(:)
!HPF$ ALIGN fac(:) WITH tplate(:)
!HPF$ ALIGN p(:,*) WITH tplate(:)
!HPF$ ALIGN q(:,*) WITH tplate(:)
C Executable Code.
      npts = NN
      WRITE(UNIT=*,FMT=101) NN
      READ  *, npts
      IF( npts .GT. NN )THEN
         WRITE(UNIT=*,FMT=102) NN
         STOP
      END IF
C Allocate arrays.
      ALLOCATE( dx(0:npts-1) )
      ALLOCATE( dy(0:npts-1) )
      ALLOCATE( dz(0:npts-1) )
      ALLOCATE( tx(0:npts-1) )
      ALLOCATE( ty(0:npts-1) )
      ALLOCATE( tz(0:npts-1) )
      ALLOCATE( sq(0:npts-1) )
      ALLOCATE( dist(0:npts-1) )
      ALLOCATE( fac(0:npts-1) )
      ALLOCATE( p(0:npts-1,0:6) )
      ALLOCATE( q(0:npts-1,0:6) )
      call timer(start,0.0)
      call timer(ti,0.0)
      tcalc = 0.0
      tcom  = 0.0
C Define inital state of particles.
      p = 0
      FORALL( i=0:npts-1 )
         p(i,MM) = 10.05 + i
         p(i,PX) = 30.0 * i
         p(i,PY) = 20.0 * i
         p(i,PZ) = 10.0 * i
      END FORALL
C  Initialize q to be the same as p.  As the force between particle i and j
C  is calculated and added into p, q will be used to accumulate the
C  symmetrical opposite force from j to i.
      q = p
      k = ((npts - 1) / 2) - 1
      call timer(tcalc,ti)
      do i = 0, k
```

```
         call timer(ti,0.0)
         q = CSHIFT(q,DIM=1,SHIFT=-1)
         call timer(a,ti)
         tcom = tcom + a
         call timer(ti,0.0)
         dx(:) = p(:,PX) - q(:,PX)
         dy(:) = p(:,PY) - q(:,PY)
         dz(:) = p(:,PZ) - q(:,PZ)
         sq(:) = dx(:)**2 + dy(:)**2 + dz(:)**2
         dist(:) = SQRT(sq(:))
         fac(:) = p(:,MM) * q(:,MM) / (dist(:) * sq(:))
         tx(:) = fac(:) * dx(:)
         ty(:) = fac(:) * dy(:)
         tz(:) = fac(:) * dz(:)
         p(:,FX) = p(:,FX) - tx(:)
         p(:,FY) = p(:,FY) - ty(:)
         p(:,FZ) = p(:,FZ) - tz(:)
         q(:,FX) = q(:,FX) + tx(:)
         q(:,FY) = q(:,FY) + ty(:)
         q(:,FZ) = q(:,FZ) + tz(:)
         call timer(a,ti)
         tcalc = tcalc + a
      end do
C Final one-way forces.
      IF( MOD(npts,2) .EQ. 0 )THEN
         CALL timer(ti,0.0)
         q = CSHIFT(q,DIM=1,SHIFT=-1)
         call timer(a,ti)
         tcom = tcom + a
         call timer(ti,0.0)
         dx(:) = p(:,PX) - q(:,PX)
         dy(:) = p(:,PY) - q(:,PY)
         dz(:) = p(:,PZ) - q(:,PZ)
         sq(:) = dx(:)**2 + dy(:)**2 + dz(:)**2
         dist(:) = SQRT(sq(:))
         fac(:) = p(:,MM) * q(:,MM) / (dist(:) * sq(:))
         tx(:) = fac(:) * dx(:)
         ty(:) = fac(:) * dy(:)
         tz(:) = fac(:) * dz(:)
         p(:,FX) = p(:,FX) - tx(:)
         p(:,FY) = p(:,FY) - ty(:)
         p(:,FZ) = p(:,FZ) - tz(:)
         call timer(a,ti)
         tcalc = tcalc + a
      END IF
C Combine final forces for the final result
C CSHIFT operation to bring q() back to original indexing state.
      call timer(ti,0.0)
      q = CSHIFT( q,DIM=1,SHIFT=(npts/2) )
      call timer(a,ti)
      tcom = tcom + a
      call timer(ti,0.0)
      p(:,FX) = p(:,FX) + q(:,FX)
      p(:,FY) = p(:,FY) + q(:,FY)
      p(:,FZ) = p(:,FZ) + q(:,FZ)
      call timer(a,ti)
      tcalc = tcalc + a
C Stop timer
      call timer(tf,start)
      j = MIN(65,npts-1)
```

```
      print *, 'State After='
      print *, 'p(FX)=',(p(i,FX),i=0,j)
      print *, 'p(FY)=',(p(i,FY),i=0,j)
      print *, 'p(FZ)=',(p(i,FZ),i=0,j)
      WRITE(6,6001) NUMBER_OF_PROCESSORS(),npts,
     1    tcom,tcalc,(tf-tcom-tcalc),tf
 101  FORMAT('Enter the number of data points - <= ',I8)
 102  FORMAT('The choice of npts=',I8,' will probably cause a crash.')
 6001 FORMAT('Number of Processors = ',I4/
     1     'Problem size = ',I5/
     2     'Communications = ',F9.3/
     3     'Compute        = ',F9.3/
     4     'Others         = ',F9.3/
     5     'Total time     = ',F9.3)
      STOP
      END
C=======================================================================
      SUBROUTINE timer(return_time, initial_time)
      real, intent(in)  :: initial_time
      real,  intent(out) :: return_time
      integer finish,rate
      CALL system_clock( COUNT=finish,COUNT_RATE=rate)
      return_time = FLOAT(finish) / FLOAT(rate) - initial_time
      RETURN
      END
C=======================================================================
```

A.10 Image Template Matching

Template Matching. This code convolves an $N \times N$ image with an $M \times M$ image. The result is stored in '`C2D()`'. A high value of '`C2D(i,j)`' will imply that the $M \times M$ subimage at '`(i,j)`' is similar to the template.

```
      PROGRAM Template_Matching
      INTEGER, PARAMETER :: N=1000,M=5
      INTEGER u,v,t1,i,j
      INTEGER C2D(0:N-1,0:N-1),IND(0:N-1,0:N-1),T(0:M-1,0:M-1)
!HPF$ ALIGN IND(I,J) WITH C2D(I,J)
!HPF$ DISTRIBUTE C2D(BLOCK,BLOCK)
C     Initialization
      FORALL(i=0:N-1,j=0:N-1) C2D(i,j) = 0
      FORALL(i=0:M-1,j=0:M-1) T(i,j)   = i+j
      FORALL(i=0:N-1,j=0:N-1) IND(i,j) = i*j
C     Performing the computation
      DO u=0,M-1
         DO v=0,M-1
            t1 = T(u,v)
            FORALL(i=0:N-1, j=0:N-1)
     $         C2D(i,j) = C2D(i,j)+t1* IND(MOD((i+u),N),MOD((j+v),N))
         END DO
      END DO
      DO i=0,N-1
         DO j=0,N-1
            PRINT *,'C2D(',i,',',j,') = ',C2D(i,j)
         END DO
      END DO
      END
```

A.11 2-Dimensional Fast Fourier Transform

The codes contained here performs a two-dimensional FFT over an example matrix —in this case a unit constant across all the elements— in the following manner:

1. Set up the input 2-D matrix.
2. Perform FFT across the leading dimension of the matrix.
3. Transform the matrix, using an F90/HPF intrinsic function.
4. Perform FFT again, across the leading dimension.
5. Compare the expected results with that obtained.

A.11.1 Main Program for the 2-D FFT Code

```
      PROGRAM constant
      IMPLICIT NONE
      INCLUDE 'fft2d.inc'
C Define interface for subroutine fft_slice.
      INTERFACE
        PURE SUBROUTINE fft_slice(a,N,isgn)
          INTEGER,INTENT(IN)                :: N,isgn
          COMPLEX,DIMENSION(N),INTENT(INOUT) :: a
        END SUBROUTINE fft_slice
      END INTERFACE
C Define interface for subroutine timer.
      INTERFACE
        SUBROUTINE timer(return_time, initial_time)
          REAL, INTENT(IN)  :: initial_time
          REAL, INTENT(OUT) :: return_time
        END SUBROUTINE timer
      END INTERFACE
      COMPLEX,DIMENSION(NN,NN) :: a
      INTEGER                  :: icol,isgn
      REAL :: ti,tf
      REAL :: tstart,tend
      REAL :: tcomms, tcomp
C Distribute array.
!HPF$ PROCESSORS procs(NUMBER_OF_PROCESSORS())
!HPF$ TEMPLATE        :: temp(NN,NN)
!HPF$ DISTRIBUTE(*,BLOCK) ONTO procs :: temp
!HPF$ ALIGN WITH temp :: a
      INTRINSIC TRANSPOSE
      INTRINSIC REAL,AIMAG,CMPLX,CABS,MAXVAL
C Begin Executable
C Start timer
      CALL timer(ti, 0.0)
      tcomp = 0.0
      tcomms = 0.0
C Set to constant 'image'.
      a  = (1.0,0.0)
C Set parameter to tell fft_slice to do FFT transform.
      isgn = 1
      CALL timer(tstart, 0.0)
```

```
!HPF$ INDEPENDENT
      DO icol=1,NN
        CALL fft_slice(a(:,icol),NN,isgn)
      END DO
      CALL timer(tend, tstart)
      tcomp = tcomp + tend
C.. Transpose
      CALL timer(tstart, 0.0)
      a = TRANSPOSE(a)
      CALL timer(tend, tstart)
      tcomms = tcomms + tend
      CALL timer(tstart, 0.0)
!HPF$ INDEPENDENT
      DO icol=1,NN
        CALL fft_slice(a(:,icol),NN,isgn)
      END DO
      CALL timer(tend, tstart)
      tcomp = tcomp + tend
C Stop timer
      CALL timer(tf, ti)
C Simple check the FFT has worked.
      WRITE(UNIT=STMOUT,FMT=1001) NN*NN
      WRITE(UNIT=STMOUT,FMT=1002) REAL(a(1,1)),AIMAG(a(1,1))
      WRITE(UNIT=STMOUT,FMT=1003) REAL(a(NN,NN)),AIMAG(a(NN,NN))
      WRITE(UNIT=*,FMT=6004) NUMBER_OF_PROCESSORS(),NN,
     1                       tcomms,tcomp,(tf-tcomms-tcomp),tf
      STOP
 1001 FORMAT('a(1,1) should be:   ',I9)
 1002 FORMAT('a(1,1) is actually: ',F12.6,F12.6)
 1003 FORMAT('a(NN,NN) should be zero.'/
     $       'a(NN,NN) is actually: ',E15.6,E15.6)
 6004 FORMAT('Number of Processors = ',I4/
     1       'Problem size = ',I5/
     2       'Communications = ',F9.3/
     3       'Compute        = ',F9.3/
     4       'Others         = ',F9.3/
     5       'Total time     = ',F9.3)
      END
C======================================================================
```

A.11.2 1-Dimensional FFT routine for the 2-D FFT Code

```
      PURE SUBROUTINE fft_slice(a,N,isgn)
      IMPLICIT NONE
C .. Arguments ..
      INTEGER,INTENT(IN)                :: N,isgn
      COMPLEX,DIMENSION(N),INTENT(INOUT) :: a
C .. HPF directives ..
C .. Local variables ..
      INTEGER         :: m,ln2,nv2,nm1
      INTEGER         :: index,jndex
      INTEGER         :: nrows,ncols,le,lev2
      INTEGER         :: ilevel,ii,jj
      COMPLEX         :: tmp,u,w
      REAL, PARAMETER :: pi=3.141592654
C .. Externals ..
      INTERFACE
        PURE INTEGER FUNCTION ilog2(n)
          INTEGER,INTENT(IN) :: n
```

```
        END FUNCTION ilog2
        PURE INTEGER FUNCTION ipow(i,j)
          INTEGER,INTENT(IN) :: i,j
        END FUNCTION ipow
      END INTERFACE
C Begin executable.
      nrows = N
      ncols = N
      nv2   = nrows/2
      nm1   = nrows - 1
      ln2   = ilog2(N)    ! Base to the 2 log of the leading dimension.
C Sort by bit-reversed ordering.
      jndex = 1
      DO index = 1,nm1
        IF( jndex .GT. index )THEN
          tmp = a(jndex) ! Swap entries
          a(jndex) = a(index)
          a(index) = tmp
        END IF
        m = nv2
        DO WHILE( (m .GE. 2) .AND. (jndex .GT. m) )
          jndex = jndex - m
          m = m / 2
        END DO
        jndex = jndex + m
      END DO
C Danielson-Lanczos algorithm for FFT.
      DO ilevel = 1,ln2
        le   = ipow(2,ilevel)
        lev2 = le / 2
        u    = CMPLX(1.0,0.0)
        w    = CMPLX(cos(isgn*pi/lev2),sin(isgn*pi/lev2))
        DO jj = 1,lev2
          DO ii = jj,nrows,le
            jndex = ii + lev2
            index = ii
            tmp   = u*a(jndex)
            a(jndex) = a(index) - tmp
            a(index) = a(index) + tmp
          END DO
          tmp = u*w
          u = tmp
        END DO
      END DO
      RETURN
      END SUBROUTINE fft_slice
C=====================================================================
```

A.12 One Dimensional Fast Fourier Transform

This code is an implementation ofa radix-2 1-D FFT. Due to the increased communications cost, it can be shown that the scalability of this algorithm is worse than for the 2-D '`TRANSPOSE()`' based algorithm given in appendix A.11, especially when there are a large number of image elements distributed to each processor.

```
      PROGRAM FFT_1DRR
```

```
      IMPLICIT NONE
C     Real FFT on 1 dimensional array
C     n is the size of the list
C     nA are the number of elements (nA = n/2)
C     n is a power of two
      INTEGER, PARAMETER :: n=1024,nA=512
      REAL  aI(0:n-1),A(0:n-1)
      INTEGER Reverse(0:nA-1)
      REAL  WI(0:nA-1),WR(0:nA-1)
      REAL  uI(0:nA-1),uR(0:NA-1)
      REAL  tI(0:nA-1),tR(0:nA-1)
      REAL  wR1,wI1,wmI,wmR
      REAL  TWOPI
      INTEGER lgn,mul,Ireverse
      INTEGER i,s,j,k,iter,MASK,partner
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
!HPF$ ALIGN aI(:) WITH A(:)
!HPF$ ALIGN Reverse(:) WITH  A(:)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
C     Initialization
      FORALL(i=0:n-1) aI(i) = i
C     Calculating log2(nA)
      i = 1
      lgn = -1
      DO WHILE (nA .ge. i)
         lgn = lgn + 1
         i = i + i
      END DO
      TWOPI = 2 * 3.1415926535897931
C     Computing Bit reverse
!HPF$ INDEPENDENT, NEW(iter)
      DO i=0,nA-1
         Reverse(i)  = 0
         iter = lgn - 1
         DO WHILE ( iter .ge. 0)
C     BTEST(i,j) should be TRUE
            IF( BTEST(i,lgn-1-iter) )THEN
               Reverse(i) = IBSET(Reverse(i),iter)
            END IF
            iter = iter - 1
         END DO
      END DO
C     Rearranging the elements
      FORALL(i=0:nA-1)
         A(Reverse(i)*2)   = aI(i*2)   ! the real part
         A(Reverse(i)*2+1) = aI(i*2+1) ! the imaginary part
      END FORALL
C     Computing wn^0, wn^1, wn^2,...,wn^(nA/2)
      wmR = cos(TWOPI/nA)
      wmI = sin(TWOPI/nA)
      wR1 = 1
      WI1 = 1
      DO j=0, (nA/2) - 1
         WI(j) = wI1
         WR(j) = WR1
         wI1 = wI1 * wmI
         wR1 = wR1 * wmR
      END DO
      iter = 1
C     Computing FFT
```

```
      mul = nA
C     The outer loop
      DO s = 1,lgn
         mul = mul / 2
         partner = iter
         MASK = iter - 1
C     BTEST(j,s-1) should be FALSE
C     The inner loop
         FORALL(j=0:nA-1, BTEST(j,s-1) .eq. .FALSE.)
            uR(j) = A(j*2)
            uI(j) = A(j*2+1)
            tR(j) = WR(IAND(MASK,j)*mul)*A((j+partner)*2)
            tI(j) = WI(IAND(MASK,j)*mul)*A((j+partner)*2+1)
            A(j*2) = uR(j) + tR(j)
            A(j*2+1) = uI(j) + tI(j)
            A((j+partner)*2) = uR(j) - tR(j)
            A((j+partner)*2+1) = uI(j) - tI(j)
         END FORALL
         iter = iter + iter
      END DO
      DO i=0,n-1
         PRINT *,' A(',i,') = ',A(i)
      END DO
      END
```

A.13 2-Dimensional Convolution

The convolution of two images 'a' and 'b' are stored in the matrix 'c', and is calculated by the steps:

1. Perform 2-D FFT for Image 'a'.
2. Perform 2-D FFT for Image 'b'.
3. Perform matrix multiplication $c = \text{FFT}(a) * \text{FFT}(b)$.
4. Perform inverse 2-D FFT on 'c' to obtain the convolution

where each 2-D FFT is a combination of parallel 1-D FFTs which are solved in the same manner as code A.11.2 above.

```
      PROGRAM convolve
      IMPLICIT NONE
      INCLUDE 'convolve.inc'
C Define interface for subroutine fftrow.
      INTERFACE
        PURE SUBROUTINE fft_slice(a,N,sign)
          INTEGER,INTENT(IN)                :: N,sign
          COMPLEX,DIMENSION(N),INTENT(INOUT) :: a
        END SUBROUTINE fft_slice
      END INTERFACE
C Define interface for subroutine timer.
      INTERFACE
        SUBROUTINE timer(return_time, initial_time)
```

```
          REAL, INTENT(IN)  :: initial_time
          REAL, INTENT(OUT) :: return_time
        END SUBROUTINE timer
      END INTERFACE
      COMPLEX, DIMENSION(NN,NN) :: a,b,c
      INTEGER                   :: icol,sign
      REAL                      :: norm
      REAL :: ti,tf
      REAL :: tstart,tend
      REAL :: tcomms, tcomp
C Distribute arrays.
!HPF$ PROCESSORS procs(NUMBER_OF_PROCESSORS())
!HPF$ TEMPLATE        :: temp(NN,NN)
!HPF$ DISTRIBUTE(*,BLOCK) ONTO procs :: temp
!HPF$ ALIGN WITH temp :: a,b,c
      INTRINSIC TRANSPOSE
      INTRINSIC REAL,AIMAG
C Begin Executable
C Start timer
      CALL timer(ti, 0.0)
      tcomp = 0.0
      tcomms = 0.0
      norm = 1.0/(NN*NN)
C Initialise the arrays.
      a = (1.0,0.0)
      b = (0.0,1.0)
C Set parameter to tell fft_slice to do FFT transform.
      sign = 1
      CALL timer(tstart, 0.0)
!HPF$ INDEPENDENT
      DO icol = 1,NN
        CALL fft_slice(a(:,icol),NN,sign)
        CALL fft_slice(b(:,icol),NN,sign)
      END DO
      CALL timer(tend, tstart)
      tcomp = tcomp + tend
C.. Transpose
      CALL timer(tstart, 0.0)
      a = TRANSPOSE(a)
      b = TRANSPOSE(b)
      CALL timer(tend, tstart)
      tcomms = tcomms + tend
      CALL timer(tstart, 0.0)
!HPF$ INDEPENDENT
      DO icol = 1,NN
        CALL fft_slice(a(:,icol),NN,sign)
        CALL fft_slice(b(:,icol),NN,sign)
      END DO
      c = a*b
C Calculate the inverse of 'c'
      sign = -1
!HPF$ INDEPENDENT
      DO icol = 1,NN
        CALL fft_slice(c(:,icol),NN,sign)
      END DO
      CALL timer(tend, tstart)
      tcomp = tcomp + tend
C.. Transpose
      CALL timer(tstart, 0.0)
      c = TRANSPOSE(c)
```

```
      CALL timer(tend, tstart)
      tcomms = tcomms + tend
      CALL timer(tstart, 0.0)
!HPF$ INDEPENDENT
      DO icol = 1,NN
        CALL fft_slice(c(:,icol),NN,sign)
      END DO
C Normalise.
      c = c * norm
      CALL timer(tend, tstart)
      tcomp = tcomp + tend
C Stop timer
      CALL timer(tf, ti)
C Simple check the FFT has worked.
      WRITE(UNIT=STMOUT,FMT=1001) NN*NN
      WRITE(UNIT=STMOUT,FMT=1002) REAL(c(1,1)),AIMAG(c(1,1))
      WRITE(UNIT=*,FMT=6004) NUMBER_OF_PROCESSORS(),NN,
     1                       tcomms,tcomp,(tf-tcomms-tcomp),tf
      STOP
 1001 FORMAT('c(1,1) should be:   ',I9)
 1002 FORMAT('c(1,1) is actually: ',E15.6,E15.6)
 6004 FORMAT('Number of Processors = ',I4/
     1       'Problem size = ',I5/
     2       'Communications = ',F9.3/
     3       'Compute        = ',F9.3/
     4       'Others         = ',F9.3/
     5       'Total time     = ',F9.3)
      END
C======================================================================
```

A.14 Hopfield Neural Network

The Hopfield neural network is a simple associative memory model with the ability to store and later retrieve a given set of images. In the version below, the code updates the state of each neuron (referred to as '`sites`' in the code) via a stochastic parallel update, with the probability of update a user-tunable parameter. Much of the computation is matrix calculations, which makes this code amenable to HPF's data parallelism execution model.

```
      PROGRAM main
      IMPLICIT NONE
      INTERFACE
        SUBROUTINE timer(return_time, initial_time)
          REAL,INTENT(IN)  :: initial_time
          REAL,INTENT(OUT) :: return_time
        END SUBROUTINE timer
      END INTERFACE
      INTEGER,PARAMETER :: STDOUT        = 6
      INTEGER,PARAMETER :: MaxIterations = 100
      REAL,PARAMETER    :: tol           = 0.99
      REAL,PARAMETER    :: updateFrac    = 1.0
      REAL :: ti,tf
      REAL :: tstart,tend
      REAL :: tcomms, tcomp
      REAL,DIMENSION(:),ALLOCATABLE   :: RealRandom
      REAL,DIMENSION(:),ALLOCATABLE   :: LocalField
      REAL,DIMENSION(:),ALLOCATABLE   :: ExternalField
```

```
      REAL,DIMENSION(:),ALLOCATABLE   :: overlap
      REAL,DIMENSION(:,:),ALLOCATABLE :: weights
      INTEGER,DIMENSION(:),ALLOCATABLE   :: sites,sites0
      INTEGER,DIMENSION(:,:),ALLOCATABLE :: patterns
      REAL :: MaxOverlap
      REAL :: NoiseFrac
      REAL :: SelfOverlap
      INTEGER :: iarg
      INTEGER :: i,iter,n,Npatterns
      CHARACTER*(128) :: argString
C ... HPF directives ...
!HPF$ PROCESSORS :: procs( NUMBER_OF_PROCESSORS() )
!HPF$ DISTRIBUTE(BLOCK) ONTO procs :: LocalField
!HPF$ ALIGN ExternalField(:) WITH LocalField(:)
!HPF$ ALIGN weights(*,:)     WITH LocalField(:)
!HPF$ ALIGN patterns(:,*)    WITH LocalField(:)
C Begin Executable
C ... Default settings (designed to loop to MaxIterations)
      tcomms = 0.0
      tcomp  = 0.0
      iarg = 0
      NoiseFrac = 0.40
      n = 256
      Npatterns = INT( (n * 0.14) + 0.5)
C PGI HPF does not implement IARGC().
      IF( iarg .GE. 1 )THEN
         CALL getarg(1,argString)
         READ(UNIT=argString,FMT=*) NoiseFrac
         WRITE(UNIT=STDOUT,FMT='(''Setting Starting Noise  = '',F12.3)')
     1         NoiseFrac
      END IF
      IF( iarg .GE. 2 )THEN
         CALL getarg(2,argString)
         READ(UNIT=argString,FMT=*) n
         WRITE(UNIT=STDOUT,FMT='(''Setting array size = '',I6)') n
         Npatterns = INT( (n * 0.14) + 0.5)
      END IF
      IF( iarg .GE. 3 )THEN
         CALL getarg(3,argString)
         READ(UNIT=argString,FMT=*) Npatterns
         WRITE(UNIT=STDOUT,FMT='(''Setting Number of patterns = '',I6)')
     1         Npatterns
      END IF
      ALLOCATE( sites(n) )
      ALLOCATE( sites0(n) )
      ALLOCATE( RealRandom(n) )
      ALLOCATE( LocalField(n) )
      ALLOCATE( ExternalField(n) )
      ALLOCATE( overlap(Npatterns) )
      ALLOCATE( weights(n,n) )
      ALLOCATE( patterns(n,Npatterns) )
C Start timer
      CALL timer(ti, 0.0)
      CALL timer(tstart,0.0)
C ... Define patterns (-1,+1) with which the network will learn.
      DO i=1,Npatterns
         CALL RANDOM_NUMBER( RealRandom )
         WHERE( RealRandom .GE. 0.5 )
            patterns(:,i) = 1.0
         ELSEWHERE
```

```
            patterns(:,i) = -1.0
         END WHERE
      END DO
C ... Hebbian learning rule for the weights ...
      weights = 0.0
      DO i=1,Npatterns
         sites = patterns(:,i)
         weights = weights + MATMUL( RESHAPE(sites,(/n,1/)),
     1                               RESHAPE(sites,(/1,n/)) )
      END DO
      weights = weights / REAL( n )
C ... Define external field ...
      ExternalField = 0.01
C ... Pick the last pattern stored as the one to retrieve.
      sites = patterns(:,Npatterns)
C ... Add some noise.
      CALL RANDOM_NUMBER( RealRandom )
      WHERE( RealRandom .LE. noiseFrac )
         sites = -sites
      END WHERE
      sites0 = sites
      overlap = RESHAPE( MATMUL( RESHAPE(sites,(/1,n/)),patterns ),
     1                   (/Npatterns/) )
      overlap = overlap / REAL( n )
      MaxOverlap = MAXVAL( overlap )
      iter = 0
      SelfOverlap = 0.0
      WRITE(UNIT=STDOUT,FMT=102) iter,SelfOverlap
      CALL timer(tend, tstart)
      tcomp = tcomp + tend
      DO WHILE( (MaxOverlap .LE. tol).AND.(iter.NE.MaxIterations) )
         iter = iter + 1
C ... Calculate the local field at each site.
         CALL timer(tstart, 0.0)
         LocalField = RESHAPE( MATMUL( RESHAPE(sites,(/1,n/)),weights ),
     1                         (/n/) ) + ExternalField
         CALL timer(tend, tstart)
         tcomms = tcomms + tend
         CALL timer(tstart, 0.0)
C ... Noisy parallel update dynamics, using sign() function.
         CALL RANDOM_NUMBER( RealRandom )
         WHERE( (LocalField.GE.0.0).AND.(RealRandom.LE.updateFrac) )
            sites = 1
         END WHERE
         WHERE( (LocalField.LT.0.0).AND.(RealRandom.LE.updateFrac) )
            sites = -1
         END WHERE
         CALL timer(tend,tstart)
         tcomp = tcomp + tend
         CALL timer(tstart, 0.0)
         overlap = RESHAPE( MATMUL( RESHAPE(sites,(/1,n/)),patterns ),
     1                      (/Npatterns/) )
         CALL timer(tend, tstart)
         tcomms = tcomms + tend
         CALL timer(tstart, 0.0)
         overlap = overlap / REAL( n )
         MaxOverlap = MAXVAL( overlap )
         SelfOverlap = DOT_PRODUCT( sites,sites0 ) / REAL(n)
         sites0 = sites
         CALL timer(tend, tstart)
```

```
         tcomp = tcomp + tend
         WRITE(UNIT=STDOUT,FMT=102) iter,SelfOverlap
      END DO
C Stop timer
      CALL timer(tf, ti)
      WRITE(UNIT=STDOUT,FMT=101) MAXLOC( overlap ), MaxOverlap
      WRITE(UNIT=*,FMT=6004) NUMBER_OF_PROCESSORS(),n,
     1                       tcomms,tcomp,(tf-tcomms-tcomp),tf
 6004 FORMAT('Number of Processors = ',I4/
     1       'Problem size = ',I5/
     2       'Communications = ',G9.3/
     3       'Compute        = ',G9.3/
     4       'Others         = ',G9.3/
     5       'Total time     = ',G9.3)
      STOP
 101  FORMAT('Pattern found is number ',I4,' with overlap = ',F12.3)
 102  FORMAT('Progress',I6,':',F12.3)
 103  FORMAT(12F9.3)
      END
C======================================================================
```

References

[1] Adams, J., Brainerd, W., Martin, J., Smith, B., and Wagener, J., *Fortran 90 Handbook: Complete ANSI/ISO Reference*, McGraw-Hill, 1991.

[2] Bailey, D., Barton, J., Lasinski, T. and Simon, H., Editors, "The NAS Parallel Benchmarks", NASA Ames, NASA Technical Memorandum 103863, July 1993.

[3] Ballard, D., and Brown, C., *Computer Vision*, Prentice-Hall, Englewood Cliffs, NJ 1982.

[4] Bozkus, Z., Choudhary, A., Fox, G., Haupt, T., and Ranka, S., "Fortran 90D/HPF compiler for distributed-memory MIMD computers: design, implementation, and performance results," *Proceedings of Supercomputing '93*, Portland, OR, 1993, p.351.

[5] Chapman, B., Mehrotra, P., Mortisch, H., and Zima, H., "Dynamic data distributions in Vienna Fortran," *Proceedings of Supercomputing '93*, Portland, OR, 1993, p.284.

[6] T. H. Cormen, C. L. Leiserson and R. L. Rivest, "Introduction to Algorithms", MIT Press/McGray Hill, 1991.

[7] Dincer, K., Fox, G.C., Hawick, K.A., "High Performance Fortran and Possible Extensions to support Conjugate Gradient Algorithms", proceedings of the 5th IEEE Int.Sym. on High Performance Distributed Computing, August 6–9 1997, Syracuse NY, pp69–77.

[8] Duff, I.S., Erisman, A.M., and Reid, J.K., "Direct Methods for Sparse Matrices", Clarendon Press, Oxford, 1985.

[9] Finucane, T., "Binomial Approximation of American Call Option Prices with Stochastic Volatilities", Journal of Finance, 1992.

[10] Fox, G.C., Hiranadani, S., Kennedy, K., Koelbel, C., Kremmer, U., Tseng, C.W., and Wu, M., "Fortran D language specification," Technical Report, Rice and Syracuse Universities, 1992.

[11] Fox, G.C., Technical Report SCCS-717, Northeast Parallel Architectures Center, `http://www.npac.syr.edu/techreports/`, June 1995.

[12] Fox, G.C., Hawick, K.A., Haupt, T., Bogucz, E. and Roe, K., "Application of High Performance Fortran", NPAC Technical Report, SCCS-727, `http://www.npac.syr.edu/techreports/`, July 1995.

[13] Fox, G.C., Williams, R.D., and Messina, P.C., *Parallel Computing Works!*, Morgan Kaufmann, San Francisco, 1994.

[14] Hawick, K.A., Yau, H.W., and Fox, G.C., "Exploiting High Performance Fortran for Computational Fluid Dynamics," *Proceedings of High-Performance Computing and Networking, May 1995, Milan Italy.*

[15] Hawick, K.A., Bell, R.S., Dickinson, A., Surry, P.D., Wylie, B.J.N., "Parallelisation of the Unified Model Data Assimilation Scheme", Invited paper, *Proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology.*

[16] Hawick, K.A., and Wallace, D.J., "High Performance Computing for Numerical Applications", Keynote address, *Proceedings of Workshop on Computational Mechanics in UK*, Association for Computational Mechanics in Engineering, Swansea, January 1993.

[17] Hebb, D.O., "The Organisation of Behavior" *New York: Wiley*, 1949.

[18] High Performance Fortran Forum (HPFF), "High Performance Fortran Language Specification," *Scientific Programming*, vol.2 no.1, July 1993. Also available on the World Wide Web via: `http://www.crpc.rice.edu/HPFF/home.html`.

[19] High Performance Fortran Forum (HPFF), "HPF-2 Scope of Activities and Motivating Applications," available on the World Wide Web via: `ftp://hpsl.cs.umd.edu/pub/hpf_bench/hpf2/hpf2.htm`, 13th of November 1994.

[20] High Performance Fortran Forum (HPFF), "High Performance Fortran v2.0β Language Specification," available on the World Wide Web via: `http://www.crpc.rice.edu/HPFF/hpf2/index.html`, 17th of August 1996.

[21] Hiranandani, S., Kennedy, K., and Tseng, C.W., "Compiler support for machine-independent parallel programming in FortranD," in *Compiler and Runtime Software for Scalable Multiprocessors*, 1991.

[22] Hiranandani, S., Kennedy, K., and Tseng, C.W., "Preliminary experiences with the FortranD compiler," *Proceedings of Supercomputing '93,* Portland, OR, 1993, p. 338.

[23] Hopfield, J.J., "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," in *Proceedings of the National Academy of Sciences, USA,* **79**, pp2554–2558, 1982.

[24] Klasky, S., Haupt, T., Fox, G.C., "Role of Message Passing HPF and DAGH in Black Hole Grand Challenge", presented to SIAM Workshop Minneapolis 'Is Message Passing Obsolete', `http://www.npac.syr.edu/users/gcf/bbhdaghhpfmar97/` 16th of March 1997.

[25] Koelbel, C.H., Loveman, D.B., Schreiber, R.S., Steele, G.L., Zosel, M.E., "The High Performance Fortran Handbook", MIT Press 1994.

[26] Kohonen, T., "Self-Organization and Associative Memory (3rd Edition)," Berlin: Springer-Verlag 1989.

[27] Kumar, V., Grama, A., Gupta, A., and Karypis, G., "Introduction to Parallel Computing: Design and Analysis of Algorithms", The Benjamin/Cummings Publishing Company, Inc, 1994.

[28] Liao, W. K., and Ranka, S., " Scalable Parallelization of the Hough Transform", *in preparation.*

[29] McCracken, N., "NPAC Educational Materials for High Performance Fortran (HPF)", `http://www.npac.syr.edu/projects/cpsedu/hpfe/`, January 1996.

[30] McMahon, M., "Benchmarking NPAC High Performance Fortran Application Kernels", `http://web.syr.edu/ mtmcmaho/`, April 1997.

[31] Makivić, M. S., "Path Integral Monte Carlo Method for Valuation of Derivative Securities: Algorithms and Parallel Implementation", *NPAC Technical Report SCCS 650*, February 1995.

[32] Mehrotra, K. G., Mohan., C. K., and Ranka, S., *Elements of Artificial Neural Networks*, MIT Press, Cambridge, Massachusetts, September 1996.

[33] Metcalf, M., Reid, J., "Fortran 90 Explained", Oxford, 1990.

[34] MPI Forum. "The Message-Passing Interface Standard," Mississippi State University `http://www.erc.msstate.edu/mpi/resources.html`, 1995.

[35] Peskin, C. and McQueen, D., "Heart Throb: Modelling Cardiac Fluid Dynamics", Pittsburgh Supercomputing Center, `http://pscinfo.psc.edu/publications/publications.html`, 1994.

[36] Presberg, D.L., "Comparisons of 3 HPF Compilers", NHSE Review `http://nhse.cs.rice.edu/NHSEreview/HPF/`, September 1996.

[37] R. Ponnusamy, J. Saltz, A. Choudhary, Yuan-Shin Hwang, and G. Fox, "Supporting Irregular Data Distributions in FORTRAN 90D/HPF Compilers," IEEE Parallel and Distributed Technology, Spring 1995.

[38] Raj, P., and Siclari, M., "Toward certifying CFD codes using wing C and M100 wing-body configurations," *Paper AIAA-94-2241* presented at 25th AIAA Fluid Dynamics Conference, Colorado Springs, CO, 20-23 June 1994.

[39] Roe, K.P., Mehrotra, P., "Implementation of a Total Variation Diminishing Scheme for the Shock Tube Problem in High Performance Fortran", presented at the Eighth SIAM Conference on Parallel Processing for Scientific Computing, March 14 1997.

[40] Sorevik, T., "Ocean Modelling code benefits from High Performance Fortran", HPCwire article number 10539, `hpcwirenewsmaster.tgc.com`, December 13th 1996.

[41] Trew, A. and Wilson, G. (eds.), "Past Present, Parallel: A Survey of Available Parallel Computing Systems", Springer-Verlag 1991.

[42] Yau, H.W., and Wallace, D.J., "Enlarging the Attractor Basins of Neural Networks with Noisy External Fields", *Journal of Physics:A Maths and General*, 24:5639–5650, 1991.

[43] Yau, H.W., Fox, G.C., and Hawick, K.A., "Evaluation of High Performance Fortran through Application Kernels", *Proceeedings of High Performance Computing and Networking 1997, Vienna, Austria*, 28-30th of April, 1997.