HPF-2 Scope of Activities and Motivating Applications

High Performance Fortran Forum

November 13, 1994
Version 0.8

Contents

Acknowledgments

This document was edited by Ian Foster, Rob Schreiber and Paul Havlak. Alok Choudhary, James Cowie, Ian Foster, Paul Havlak, Chuck Koelbel, Piyush Mehrotra, Andy Meltzer, and Rob Schreiber contributed sections. Gail Pieper provided editorial assistance. Many members of the HPF Forum contributed the applications, as noted for each of them in Section 4.

Section 1

Executive Summary

This document presents issues recommended by the HPF Forum for consideration under the HPF-2 effort. After this summary, it comprises three parts: a set of proposed new capabilities, a proposal for a *Kernel HPF*—an official subset designed for particularly high performance—and a set of applications motivating the new capabilities.

The proposed new capabilities are grouped into general areas, as follows.

Enhanced Mapping:

1. **Irregular mapping of arrays.** Arbitrary, user-specified mapping of array elements to members of abstract processor arrangements, allowing the specification of the mapping through *map* arrays or mapping functions, and the mapping of arrays to processor arrangements of lower rank.

2. **Mapping of linked data structures.** Dynamic, user-specified mapping of scalars (especially scalars of derived type that form cells in linked structures) to individual elements of a processor arrangement.

3. **Mapping of derived type components.** Mapping of components of unmapped instances of scalars of derived type, and mapping of components in a derived type definition.

4. **Mapping to processor subsets.** Allowing the target processor arrangement in a `DISTRIBUTE` directive to be a section of a processors arrangement, and allowing the distribution of dummy array arguments onto sections of processor arrangements and onto undersized processor arrangements.

5. **Processor views.** Allowing multiple views of a processor arrangement.

6. **Partial replication to processor subsets.** User-specified mapping of array elements and scalars to an arbitrary processor subset.

Computation Control and Task Parallelism:

1. **Computation mapping.** The ability to control the mapping of computation to processors.

2. **Reductions and atomic operations.** Allowing iterations of an `INDEPENDENT DO` loop to affect each other by shared variables or to otherwise combine results.

3. **DOACROSS loops.** The ability to specify that loop iterations should be executed when some synchronization condition is satisfied.

4. **Multiple HPF process model.** The ability to explicitly start HPF processes on processor subsets, terminate HPF processes, and communicate between HPF processes (i.e., task parallelism).

5. **Parallel sections.** Means of delineating and synchronizing parallel sections, as for example in PCF Fortran.

Input-Output:

1. **Transparent parallel accesses to files.** Extensions to `OPEN` that allow the creation of files that will be accessed in a restricted manner, to allow faster parallel I/O.

2. **Striping/distributing files over disks.** Directives or extensions to `OPEN` for optimizing file mapping to multiple disk systems. Library routines for obtaining disk system parameters.

3. **Out-of-core and persistent arrays.** Directives to advise that an array should be demand paged and to advise on the characteristics of the page.

4. **Checkpointing/Restart.** A checkpoint statement with corresponding restart capability.

5. **Asynchronous I/O and prefetching.** Split-phase `READ` and `WRITE`.

Communication Optimizations:

1. **Asserting the reuse of a communication pattern.**

2. **Split-phase dereference of distributed objects.**

3. **Locality assertions.** Methods for advising that the iterations assigned to a given set of processors reference only data mapped to those processors.

4. **Distribution views.** Methods for providing multiple views of the same distributed data.

Language Processor Environment:

1. **Language interoperability.** Standard interfaces that allow HPF to call other languages, and other languages to call HPF.

2. **Tool support.** Standard interfaces for use by debuggers, profilers, and other tools.

Section 2

Justification for New Capabilities

We provide technical justification for each new capability outlined in the Executive Summary (with reference to motivating applications) and describe what is known about implementation techniques.

2.1 Enhanced Mapping

Although HPF-1 provides a variety of data mapping options, all produce regular patterns. Many in the community argue that these are not sufficient for programming (at least some) complex algorithms. In particular, the following algorithmic features tend to require more complex partitioning of data:

- **Unbalanced load.** When the work required by different elements varies, it may be advantageous to balance the computational load by distributing the elements unevenly. HPF-1 distributions can become unbalanced if the number of elements is not divisible by the number of processors, but this is considered neither desirable nor elegant for this purpose.

- **Unstructured data access.** When the data structure has complex interconnections, it may be important to place sections that are "near" each other on the same processor. This arrangement can create sections with somewhat irregular boundaries, rather than the orthogonal cutting planes derived from HPF-1 distributions; the problem can be especially severe when the data structure indexing is not related to the connectivity.

- **Nonlinear (pointer-based) data structures.** When there is no analog of array indices, it is difficult to succinctly describe any alignment between the structures. For similar reasons, the lack of an *a priori* organization (that is, the fact that the data structure is not an array) makes it hard to describe a general distribution pattern.

- **Dynamically-built data structures.** When the data structure is built incrementally, it is probably inappropriate (not to mention confusing) to describe the data mapping before the structure exists. Instead, it may be better to describe the mapping of each piece as it is added. Alternatively, one may wish to delay any mapping until the data structure is completed.

No single distribution method (or even framework) has yet been suggested that solves all these problems efficiently. However, many researchers have suggested partial solutions. The sections below describe some of these ideas as they might be applied to HPF.

2.1.1 Irregular Mapping of Arrays

To solve the problems of unbalanced load and unstructured data access of arrays, researchers have described many new distribution patterns. A few generalizations of HPF-1 alignments have also been suggested. We first note several issues that are common to many of these mappings, and then discuss some of the better-known approaches.

Designers of new data distributions and alignments generally assume that reducing the volume of data communicated is paramount to improving parallel performance. To achieve this, they map data items that "interact" with each other onto the same processor as much as possible. For example, in a molecular dynamics simulation, atoms that are bonded to each other would be mapped to the same processor. It is up to the compiler to match the computation mapping to the data mapping. This matching is usually done by some variant of the "owner-computes rule," in which the processor that owns one reference in a statement performs the entire computation of that statement. Alternatively, the processor to perform a computation may be specified directly as discussed in Section 2.2.1. The new distributions generally involve more complex address calculation formulas than do HPF-1 patterns; it is assumed (and sometimes checked experimentally) that this cost is overshadowed by the benefits of the new distribution.

Generalized Block Distributions. The simplest new form of mapping simply extends the HPF-1 `BLOCK` distribution to allow different block sizes on each processor. This idea was discussed by Fox [42] and was implemented as part of the Superb environment [46, 103] and later in the Vienna Fortran Compiler System. Similar features appear in several language descriptions, but to our knowledge no performance results have been reported from any implementation. There are no obvious implementation difficulties, however. A slightly different idea was used in the Paragon compiler, which allowed partitioning of 2-D arrays by arbitrary-sized rectangles [78]. Both the one-dimensional and multidimensional versions have many of the advantages of simple `BLOCK` distribution for nearest-neighbor communication and are useful for solving load-balancing problems. In addition, the addressing formulas are fairly simple; the most complex operation is looking up the processor owning an element (which requires some form of search, rather than the closed-form expressions for `BLOCK` and `CYCLIC`). Generalizing the one-dimensional `BLOCK` pattern seems to be a relatively small change to HPF.

Map Arrays. Perhaps at the opposite end of the complexity scale are map arrays. These distributions use another array (that may itself be distributed) to record the mapping from array elements to processors. This form of distribution was popularized by Saltz and his coworkers [83] and has appeared in Fortran D [41] and Vienna Fortran [19]. A number of papers have presented performance results for programs that use these distributions [16, 24, 83]. It is clear that substantial compiler optimization is necessary for acceptable performance. The primary implementation technique is the inspector-executor paradigm, which divides communication into a setup phase (the inspector) and the actual data communication (the executor). Further optimizations are needed to avoid explicit index computations. Map arrays can represent any many-to-one mapping of elements to processors; of course, the performance of the program will depend on which mapping is actually used. Hence, this approach provides great flexibility but also introduces the danger that an extremely bad mapping may be specified accidentally. Adding these distributions to HPF would add much power, but at the expense of great implementation difficulty. (But note that much of this difficulty is present in the implementation of vector-valued subscripts in HPF.)

User-defined Distribution Functions. Allowing users to define distribution functions directly is another approach to providing fully general distributions. The principle is identical to map arrays, except that the mapping is defined by a function rather than an array. This idea was implicit in Id Nouveau [80] and Kali [59] and also appeared in Vienna Fortran [19]. The details of what a distribution function should return varied between languages; some required only the processor number, while others also had the function specify the on-processor addressing. None of the groups reported performance results using distribution functions, but one might expect the implementation to follow lines similar to map arrays. (It is noteworthy that distribution functions potentially use much less memory that map arrays.) Adding these distributions has advantages and pitfalls similar to those associated with map arrays.

Value-based Partitioning. There have been several implementations of implicitly specified or value-based partitioning. In contrast to the above *explicit* specifications of irregular data distributions, the user specifies the type of information to be used in data partitioning as well as the irregular data partitioning heuristic to be used. Language extensions have been designed and implemented to allow users to specify the information needed to produce an irregular distribution. Based on user directives, the compiler produces code that, at runtime, passes the user-specified partitioning information to a (user specified) partitioner. This approach to partitioning has recently been implemented in the Fortran 77D compiler [50], employing the CHAOS runtime system [26, 75].

Recent experience has indicated that users can, with some effort, do without irregular distribution. An irregular mapping can be embedded in an HPF regular mapping by reordering elements of data arrays and renumbering indirection arrays. This may yield performance comparable to that achieved by a compiler for a language (such as Fortran 90D) that directly supports irregular distributions [74]. In order to use the reordering method, however, users are forced to make numerous calls to extrinsic library functions.

Although less work has been done on extending alignment, many of the distribution extensions above have counterparts there. Much research has been done on skewed alignments, which extend HPF-1's affine alignment functions to more general linear functions [4, 76]. Often the alignments are generated automatically based on program analysis. Interesting results have been obtained for single loop nests, but it remains to be seen whether these can be extended to full programs. An obvious additional extension is to allow general expressions in `ALIGN` directives. The problems with implementing this option are similar to the problems of implementing map arrays or distribution functions. However, the fully general alignments are quite useful for "connecting" different classes of data elements, such as the vertices and edges in an unstructured mesh.

It is also worth noting that many groups are working on automatic distribution of data structures, either in regular patterns or by creating more general patterns. This is an area of active research, and there is little consensus on the best approach(es) to take. Therefore, the implications for HPF are somewhat unclear.

2.1.2 Mapping of Linked Data Structures

To handle the problems of nonlinear and dynamically-built data structures, researchers have proposed several pointer mapping mechanisms [17, 97]. Like map arrays, these methods must be coupled with a rational partitioning of data. In the HPF context, the most pressing need is to map scalars of derived types, which are referred to by pointers and are dynamically allocated. For example, in the process of building a tree, each node would be assigned to a processor (using `ALIGN` or `DISTRIBUTE`) when it is allocated. If the tree needed remapping later, a walk over the tree could be used to relocate the nodes (using `REALIGN` or `REDISTRIBUTE`). This is already possible in HPF-1 by using somewhat baroque syntax (effectively, the new nodes must appear in a `REALIGN` immediately after they are allocated). However, a more convenient method is needed to promote the use of efficient advanced data structures. This should be carefully designed to allow the structure to be built in parallel as well; doing so requires interaction with the features discussed in other sections.

Whatever mechanism is used for mapping the data structure, implementing the program efficiently remains a significant challenge. Approaches to date rely on the inspector-executor paradigm or on lightweight communications mechanisms. The basic techniques require representing a pointer as a processor and location pair. Maintaining this representation when the underlying objects can be remapped requires an additional protocol, usually implemented in the runtime system. The performance implication of this support is unclear, as is the difficulty of incorporating it in a commercial compiler.

In summary, it is important to allow pointer-based data structures to be partitioned among processors. The implementation costs of this feature are not well understood, however.

2.1.3 Mapping of Derived Type Components

HPF-1 does not allow components of derived types to be mapped directly. This can be a limitation for programs that store several grids in a single structure. For example, if a derived type contains a large (fixed-size) grid as a component, then the user would almost certainly want to distribute it. In this case, adding the distribution to the derived type declaration would be appropriate. Other applications might use a pointer component to an array, thus allowing the array size to vary between derived type instances. In this case, efficiency may require different mappings for each instance. Alternately, it may be more convenient to specify the mapping for the corresponding component in every instance of a derived type. Benkner [11] discusses both possibilities.

Although we do not know of any direct implementation experience in this area, it is easy to imagine how it could be done. If the mapping information is part of the derived type, the techniques used for ordinary variables can be employed. Memory allocation for the components is somewhat challenging if `REDISTRIBUTE` is allowed, but this difficulty can be overcome by using pointers in the underlying implementation. In the more dynamic case, the techniques already needed for tracking pointers to mapped arrays can be applied to pointer components as well. Here we assume that the instance of the derived type containing the mapped array is not itself explicitly mapped. The meaning of such a multilevel mapping is not entirely clear.

It appears straightforward to add both of these possibilities to HPF. In fact, it can be done indirectly already using pointer components and `ALLOCATABLE` arrays. However, a simpler syntax would be much appreciated.

2.1.4 Mapping to Processor Subsets

HPF-1 forces every template to be distributed over all processors. Some programs can profitably use subsets of processors. For example, in a multigrid application the finest mesh level may fill the entire processor array but the coarsest mesh may be much smaller; spreading the coarse mesh across many processors may generate too much fine-grain communication for efficiency. Another example is task-parallel computations; if only one task accesses an array, the array should only be stored on processors executing that task. While it is possible to achieve some of these "subset mappings" by aligning an array to a much larger template, such programming is inconvenient and may defeat the compiler's analysis. Instead, directly specifying a subset of processors in the `DISTRIBUTE` directive is preferable. This is possible in Vienna Fortran [19], and a proposal based on that capability appeared in the HPF *Journal of Development* [54]. Modest performance gains for a block-structured Navier-Stokes code due to mapping to subsets have also been reported [2, 3].

To our knowledge, no detailed description of an implementation of processor subset distributions has been published. A reasonable strategy for such an implementation, however, would be to use symbolic quantities in the compiler for parameters such as the number of processors and the current processor. In a sense, the compiler would act as if the distribution were over the entire machine, but processors not involved in a particular mapping would skip those sections of code. Interactions between arrays distributed to different processor sets would require careful management. What effect either the symbolic computations or the extra compilation technology would have on efficiency is unclear.

One special case of processor subset mappings is both important and simpler than the above discussion. When an array section is passed to a procedure, the section may already be restricted to a subset of processors. Aligning local arrays with the parameters effectively propagates this subset mapping. As long as the subroutine accessed no global data, it would only "see" the processors that passed the arguments. Descriptive mappings could reasonably refer to this set of active processors, and the compilation (using symbolics) would pose few problems. Alternately, it might be natural for a descriptive mapping to refer to a section of a processor array, thus providing more information to the compiler.

To summarize, mapping to processor subsets is vital for certain problems. Such mappings are already performed in programs written in lower-level languages. However, generating such mappings from high-level languages is not a solved problem. Restricting such mappings to procedure arguments may simplify these problems.

2.1.5 Processor Views

HPF-1 directives do not allow an array of rank k to be directly distributed to an abstract processor set of rank greater than k. Thus, if we have a two-dimensional matrix distributed across a two-dimensional set of abstract processors, there is no mechanism for distributing a one-dimensional vector across the same set of processors. In other words, there is no way to view the same set of abstract processors as a processor array having a different shape, in particular a different rank.

The HPF *Journal of Development* [54] includes a proposal for the `VIEW` directive which allowed processor arrangements to be "equivalenced" using Fortran column-major ordering thus allowing the same set of abstract processors to be viewed as having different rectilinear geometries. Another proposal would allow processor arrays to be aligned to each other using the syntax of the `ALIGN` directive. In this case, two abstract processors aligned to each other would be mapped by the compiler to the same physical processor. This allows more flexibility, since the user can use any linear mapping allowed by the align syntax to equivalence the processor arrangements; but it does not allow the change of rank discussed above. Also, this would allow processor subsets to be easily specified, since a smaller array of processors could be aligned to a larger array, thus providing a name for a subset of the larger array.

Implementation of processor alignments would introduce another level of mapping; this, however, could easily be collapsed into the function mapping abstract to physical processors.

2.1.6 Partial Replication to Processor Subsets

An important optimization for improving communication performance in an HPF program's execution is to break the execution into a sequence of phases. At the end of each phase, in preparation for the following phase, all data referenced by a processor is communicated to that processor, so that the following computation will occur without communication, and so that messages will be as large as possible. Call the data imported to a processor the "locally essential" portion of the global, distributed data structures.

In the case of a vector-valued subscript, for example, the set of locally essential data can be found before a phase begins. But consider the following loop.

```
!HPF$ INDEPENDENT, NEW V
        DO K = 1, M
           V = VO(K)
           DO J = 1, N
             A(K) + A(K) + B(V)
             V = USER_FN( K, J, V )
           ENDDO
        ENDDO
```

Here the reference pattern is data dependent and cannot be known by the compiler prior to execution of the loop. In some instances (see the Barnes-Hut application described in Section 4.1) the user may be able to specify the set of data used by each processor, or, equivalently, for each data element (of the array B in the example above) the set of processors that will (or may) use it. Suppose the latter. The user computes READER(I,J) and NUM_READERS(I) such that B(I) will be referenced only by processors READER(I,J), $1 \leq$ J $\leq$ NUM_READERS(I). We then need a means to assert that B should be copied to these processors and that references to the global B should be remapped to the local copy.

2.2 Computation Control and Task Parallelism

HPF-1's computational model emphasizes support for fine-grained data parallelism, in which DO loops and other constructs are used to expose parallelism at the level of individual data points. (The EXTRINSIC mechanism is the one exception to this rule.) The user has no direct control over the mapping or scheduling of these fine-grained units of computation; these tasks are the responsibility of the compiler.

Some interesting applications appear to require either user control over mapping and scheduling and/or the ability to specify more coarse-grained parallelism. Motivations for these features include the following:

- **Software engineering.** Some parallel computations are naturally described in terms of communicating, coarse-grained tasks; implementations with the same structure may be simpler and more modular than purely data-parallel programs.

- **Concurrency.** A user may know that program components other than DO loop iterations can execute concurrently: for example, recursive calls in a divide-and-conquer algorithm or successive stages in an image-processing pipeline. While this sort of concurrency can sometimes be expressed in HPF-1 (e.g., see the Quicksort procedure in Section 2.4.3), the resulting programs are neither elegant nor easily compiled.

- **Scheduling.** Iterations of a DO loop may be able to execute concurrently if certain scheduling constraints, known to the user but not easily determined by the compiler, are satisfied. The user should be able to provide this information.

No single framework has been proposed that addresses all of these issues efficiently. However, many researchers have suggested partial solutions. In the rest of this section, we describe some of these ideas as they might be applied to HPF. In the first three subsections, we describe extensions that can be used to control the mapping or scheduling of `DO` loop iterations, while preserving sequential semantics. In the fourth and fifth subsections, we describe extensions that introduce explicit tasking. In discussing the latter extensions, we shall use the term *process* to denote a parallel computation comprising one or more logical *threads* running in a separate *name space*, corresponding to a specified HPF program, with its associated subroutines, modules, files, etc. As we shall see, both processes and threads may be created by a parallel section construct or by some form of asynchronous subroutine call, or *spawn*, that returns control to the caller while the callee runs concurrently. Threads of the same process can share global variables through ordinary Fortran 90 scoping (e.g., through modules). On the other hand, threads executing in different processes have no way to share access to the same global variables and need some new mechanism to communicate.

A process may, of course, execute on multiple HPF processors, and may distribute the data in its name space over them using HPF data mapping language. The processors of two different processes are conceptually different and have nothing in common. That fact does not preclude the use of overlapping sets of physical processors or of disjoint machines connected by E-mail, for example, but we view these as orthogonal issues.

Threads in the same process execute on the same processors. In particular, they can share an HPF `PROCESSORS` arrangement via a module. `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE` return the same result on all the threads of a process.

An HPF-1 program is a process consisting of one thread. While it may use `FORALL` and `INDEPENDENT` to express parallel computations that are irregular, its individual loop iterations are not what we call threads because, unlike threads, they cannot communicate during execution. More important, they are not loci of control: the compiler may choose to distribute them to any number of actual threads in some unspecified manner.

2.2.1 Computation Mapping

Distribution directives allow the user to control the mapping of data elements to the memories of the underlying system. There is no corresponding support for mapping the computation to specific processors. In many cases, the compiler with enough analysis can determine the "best" processor for executing a given computation. However, this determination may not be possible in the presence of complex code, for example when mapping an iteration of an `INDEPENDENT` loop in the presence of indirect array accesses. In such situations, allowing the user to specify the mapping of computation to the processors can result in better load balance while reducing communication costs.

The concept of using an `ON` clause to map parallel iterations was first introduced by Kali and later adopted by Fortran D and Vienna Fortran. This clause allowed the user to specify the processor on which an iteration of the parallel loop should be executed, either directly by specifying an element of the processor array or indirectly through the ownership of the specified array element. A similar concept, called the `EXECUTE_ON_HOME` directive, is proposed in the HPF *Journal of Development* [54]. However, this coupling of the concept of computation mapping (`EXECUTE_ON`) with a mapping from data elements to processors (`HOME`) is unfortunate, because both those ideas are potentially valuable independently.

The `EXECUTE_ON` directive could be applied to specify the mapping of `INDEPENDENT DO` loops, `FORALL` constructs and statements, and other indexed array assignments statements. A *local-access-directive* could be used to assert that that elements of the specified arrays accessed in an iteration are local if the specified mapping of iterations to processors is enforced; this is discussed further in Section 2.4.3.

The EXECUTE_ON directive can be further extended in several ways. First it could be used with other statements; for example, if applied to a subroutine call statement, it could allow the user to control where the subroutine should be executed. Second, instead of specifying a single processor, the directive could specify a subset of processors to be used for the execution of the associated statement. Thus, for example, in a double nested loop, each of the iterations of the outer loop could be mapped to a "row" of a two-dimensional array of processors, while the inner iteration could be spread across the associated row. Similarly, a subroutine call could be mapped to a subset of processors where the distribution of the computation across this subset is controlled from within the subroutine itself.

These concepts allow the user to control the mapping of computation to the underlying processor set. This may result in better load balance and reduced communication.

2.2.2 Reductions and Atomic Operations

The HPF INDEPENDENT directive asserts that no iteration of a DO loop can affect any other iteration either directly or indirectly. One consequence of this restriction is that no scalar variable can be modified in more than one iteration of an independent loop unless the variable has been declared to be NEW. Thus, for example, one cannot accumulate values generated by the iterations into a single scalar variable. The values can be stored in an array and then accumulated after the loop, but this approach does not seem natural and also implies that storage proportional to the size of the iteration space has been allocated. (This is impractical in the Fock matrix application of Section 4.11, for example). We describe three approaches to this problem, of increasing generality (and complexity): reduction operators, user-defined reduction functions, and critical sections.

Reductions have been supported in research compilers with some success, both for irregular summations and for list building [102, 85]. They are part of Fortran D and the MPI standard [41, 37].

Reduction Operators. A REDUCTION directive can be placed immediately before an assignment statement to indicate that the statement represents a reduction across the iterations. The assignment statement is called a *reduction assignment* and is restricted to be of the following form:

```
variable = variable op expr
```

The variable, called the *reduction variable*, is the target variable into which the values from all the iterations are accumulated. The effect of the loop is to accumulate into the target variable the values of the expression expr that arise from the different loop iterations, using the specified operator op and combining the values with the initial value of the reduction variable. The following example illustrates the use of a REDUCTION directive to sum values of a distributed array in an INDEPENDENT loop.

```
       X = 3.1415926
!HPF$  INDEPENDENT
       DO I = 1, N
         ...
!HPF2$   REDUCTION
         X = X - A(I)
         ...
       ENDDO
!  FINAL X is 3.1415926 - SUM(A(1:N))
```

Any binary operator is allowed, including user-defined and overloaded intrinsic operators.

The REDUCTION directive allows the compiler to reorder and reassociate invocations of the operator. By using this directive, the user asserts that the computed result is the same as it would be without the reduction and independent directives (or that the user does not care about the difference). In other words, the loop computes the same result as in Fortran 90. Without the reduction directive, it would be incorrect to use the INDEPENDENT directive for the loop. Note that the final value of the reduction variable is not available until the end of the loop.

If the target variable is an array, different iterations may accumulate into different elements of the array, as shown in the following code fragment extracted from an unstructured mesh example.

```
!HPF$  INDEPENDENT
       DO I = 1, NEDGES
          J1 = EDGE(I, 1)
          J2 = EDGE(I, 2)
          ...
!HPF2$    REDUCTION
          X(J1)  = X(J1)  + value
!HPF2$    REDUCTION
          X(J2)  = X(J2)  - value
          ...
       ENDDO
```

We propose some further semantic requirements here. A reduction variable may not appear in any context in the loop body except in the two indicated positions in a reduction assignment. The reduction variable may be a scalar, an array, an array element, or an array section. If a variable, or an element or section of an array variable occurs as a reduction variable, then no instance of this variable may occur elsewhere in the loop except in reduction assignments. A variable, array element, or array section may be used as the target of multiple reduction assignments within a loop, as long as the same reduction operator is used in every such assignment involving the given variable. However, for purposes of this restriction, the addition (+) and the subtraction (−) operators are considered to be the "same reduction operator," as are the multiplication (*) and division (/) operators.

A reasonable implementation of reductions uses a local (one-per-processor) accumulator variable into which all reduction operations are applied without interprocessor communication, with a global accumulation of the final value after the execution of all loop iterations. The local accumulators must be initialized not to the initial value of the reduction variable (π in the example) but to the identity element of the reduction operation. For reductions involving intrinsic operators, the compiler has this value. For other reductions, the user must supply it. Some syntactic extension to the REDUCTION directive would be needed, for example,

```
!HPF2$   REDUCTION, IDENTITY = <expr>
```

User-defined Reduction Functions. The form of the reduction operation can be generalized to allow user-defined functions by permitting assignment statements of the following form after the REDUCTION directive.

```
           variable = function-name(variable, expr)
```

Here, the function `function-name` is restricted to be a function of two arguments, and the first one must be the same reduction variable reference that occurs on the left-hand side. The semantics of the statement and the restrictions are the same as in the case of reduction operators.

Examples of user-defined reductions are list operations used to aggregate data items moving between processors in the Binz vortex code and discrete simulation Monte Carlo (see Sections 4.7 and 4.8).

Critical Sections. More general kinds of updates to shared variables in parallel loops, which would inhibit the use of `INDEPENDENT`, can be accommodated in these loops through the use of critical sections, as defined in PCF Fortran. The construct

```
!HPF2$  CRITICAL
!HPF2$  END CRITICAL
```

delimits a critical section: a code sequence that can be executed by only one iteration at a time. Similarly, the construct

```
!HPF2$  CRITICAL (i)(n)
!HPF2$  END CRITICAL
```

delimits a critical section protected by the ith out of n locks that are specific to this section. The direct-simulation Monte Carlo application described in Section 4.8 illustrates the use of the latter construct. This uses one lock per cell to guard the addition and deletion of particles to the per-cell lists. An iteration excludes from concurrent entry into the critical section only other iterations that are updating the same cell. Arrays of named locks are another possible extension to this concept.

An advantage of the critical section is that it is a relatively well understood, general mechanism. It enables parallel execution of a very wide class of `DO` loops, allowing arbitrary updating of shared variables within such a loop. A disadvantage is that it may be *too* general. Critical sections can be used to implement a wide variety of behavior. The performance on NUMA machines of programs that do fine-grained updates to such shared objects may be a serious problem.

Critical sections are not an ideal mechanism for implementing reductions. A compiler may be able to detect the accumulator and the operator inside a critical section, but in general it will be impossible to determine automatically that the operator is commutative and associative, and hence to employ the implementation techniques described above. Hence, critical sections are not an alternative to reductions.

2.2.3 DOACROSS Loops

In what are called *doacross loops*, loop-carried dependencies that prevent a loop from being characterized as `INDEPENDENT` can be respected via synchronization operations. One possible mechanism that allows the user to control this synchronization at a high level is the following `DOACROSS` directive:

```
          !HPF2$  DOACROSS WHEN ( logical-expr )
```

This directive must occur immediately before the associated `DO` loop. The logical expression, which contains the index variable of the loop, controls the execution of the indicated iteration. For example, in the following loop

```
LOGICAL, DIMENSION(N) :: READY
REAL   , DIMENSION(N) :: A

A(1) = 22. / 7.
READY(1) = .true.
 !HPF2$ DOACROSS WHEN ( READY(I) )
        DO I = 2, N
          ...
          A(I) = FUN(A(I-1), ...)
          READY(I) = .true.
          CALL ODIOUS_BITBASHING
        ENDDO
```

iteration I is executed only when the logical array element READY(I) is true. The user is responsible for initializing the READY array and for resetting the elements to .TRUE. when the appropriate dependence has been satisfied.

2.2.4 Multiple HPF Process Model

This subsection is concerned with mechanisms that allow the creation of multiple HPF processes, which interact by using various mechanisms not supported in HPF-1. Note that HPF-1's EXTRINSIC function interface provides one such mechanism: a call to an extrinsic function creates one coarse-grained process per physical processor, and these processes can then interact by message passing or some other mechanism. However, this facility is not felt to be sufficiently general. For example, it does not allow the user to specify that a computation comprises two communicating HPF-1 programs.

A wide variety of schemes have been proposed and explored for creating processes and for exchanging data between processes [10]. For example, processes can be created by using an unstructured spawn construct or by using a structured parallel block or DO loop. Communication can be achieved by using message passing, channels or virtual files, tuple space, or monitors. Processes may be relatively coarse-grained objects (appropriate for multidisciplinary simulations, for example) or quite fine-grained objects (useful, for example, in divide-and-conquer applications). There is considerable experience with all of these approaches, but little experience with their use in an HPF context. Exceptions are the work on Fortran M [38, 39] and Opus [20].

A multiple HPF process model may allow multiple threads per name space (using a parallel section construct, to be defined in the next subsection) or may restrict programs to just one thread per name space. In either case, the existence of multiple name spaces (processes) provides a high degree of modularity, which can have software engineering advantages. As each process encapsulates its own program and data, we do not need to be concerned with name space clashes. There is also less chance of race conditions, since interactions between processes must be specified explicitly rather than occurring implicitly via data sharing. In the following, we review some possible interaction mechanisms.

The ability to create multiple HPF processes requires a variety of extensions to HPF-1 compiler and runtime system techniques. Many of the issues that arise are discussed in [18]. For example, as when compiling for processor subsets, one can use symbolic quantities in the compiler for parameters such as the number of processors and the current processor [38]. The compiler can then compile each process as if it were to execute on the entire machine.

Virtual Channels/Files. Processes can communicate by sending and receiving messages on explicitly created virtual channels [39]. These virtual channels can be represented as *virtual files*, in which case existing Fortran I/O operations (extended with appropriate keywords) can be used to create, open, write, and read virtual files, check for pending messages, etc. Different access modes can be defined that allow for multiple writers and multiple readers. For example, an OPEN call may specify that a file is a virtual file (`OPEN(mode=virtual, ...)`; that it supports multiple or single writers (`writers=multiple`, `writers=single`); that it supports multiple or single readers (`readers=multiple`, `readers=single`); and that it associates a separate file pointer with each reader, so that each reader reads all records, or that it associates a single file pointer with the file, so that each read returns a new record (`readtype=dependent`, `readtype=independent`). Read operations would normally be required to block until data is available.

Advantages of this approach are that it builds on existing concepts and I/O libraries (in particular, communications between data-parallel tasks can use the same concepts as HPF file I/O), race conditions are easily avoided (just ensure that each file has one writer and one reader), and it naturally meshes with extensions for multiple-process I/O. A disadvantage is that dynamic interaction patterns can require complex "plumbing."

Message Passing. Alternatively, HPF processes can interact by sending and receiving messages rather than performing virtual file operations. Many of the concepts developed in the MPI message passing standard can then be used. Note that communications are between HPF processes, not processors; hence, communications can involve multiple processors and the remapping of distributed data structures, if send and receive operations are applied to distributed arrays. Note also that the usual system inquiry functions `MYNODE` and `NUMNODES` would have to be renamed, as they return process and not processor statistics.

Shared Data Abstraction. Yet another interaction mechanism, proposed by Chapman et al. [20], uses shared modules with access controlled by a monitor mechanism. A form of remote procedure call is used to operate on data in the shared module; the monitor mechanism ensures mutual exclusion of concurrent RPCs to the same module. Note that there are some similarities to the Linda tuple space.

An advantage of SDAs is that it provides an arm's-length interaction model, which enhances modularity. This is particularly good for multidisciplinary applications. It appears less well suited for fine-grained or communication-intensive applications, because of the use of an intermediate space. For example, a data transfer involves two calls, and a program that needs to "probe" for incoming data must make repeated RPCs.

2.2.5 Parallel Sections

In an alternative explicitly parallel model, coarse-grained threads are created explicitly within a single name space. Thread creation can be achieved using a parallel section construct. For example, the following syntax might be used to create one thread of control for each function call `F1`, ..., `Fn`.

```
!HPF2$  BEGIN PARALLEL
        CALL F1
        ...
        CALL Fn
!HPF2$  END PARALLEL
```

As threads execute within a single name space, they can interact simply by reading and writing shared data objects. Access to shared data may be controlled by locks, barriers, etc., as in the X3H5 (PCF Fortran) model [101]. A more restrictive model, proposed by Subhlok et al. [88], allows the user to provide def-use information to synchronize process execution.

An advantage of this approach is that existing programs can be parallelized relatively easily. In cases in which a shared data structure plays a central role, it seems natural to use this model. Disadvantages are that modularity is hard to achieve (lack of scoping) and that (in the PCF model) race conditions are easy to introduce.

Parallel sections can be implemented quite naturally on shared memory hardware. On distributed memory hardware, they probably require the use of a virtual shared memory software layer. Whether and how well this works is currently a research problem. The alternative, which is to have fixed, user-specified, or compiler-specified locations for data objects with no local caching, is likely to fail for Barnes-Hut (4.1) and other important motivating applications.

A simpler proposal would provide a restricted parallel section construct as a syntactically more elegant alternative for the following code.

```
!HPF$ INDEPENDENT
       DO I = 1, 3
         SELECT CASE (I)
           CASE (1)
              <first section>
           CASE (2)
              <second section>
           CASE (3)
              <third section>
         END SELECT
       ENDDO
```

Here, there can be no interaction or communication between the parallel regions of code, except as allowed by the critical sections described in Section 2.2.2 and the doacross loops described in Section 2.2.3.

2.3 Input-Output

Incorporating features in HPF to allow high-performance input-output is considered important in order to provide scalable performance for accessing secondary storage and beyond. Several basic requirements have been identified:

- transparent parallel accesses to files,
- out-of-core arrays and persistent arrays,
- checkpointing and restart,
- the ability to stripe files over disks, and
- asynchronous I/O for overlapping computation with I/O.

Of course, many of these issues are also important in the context of Fortran 90 and may be more appropriately addressed in the context of Fortran 95 or Fortran 2000. Where appropriate, we comment on those issues that appear particularly germane to HPF.

2.3.1 Transparent Parallel Accesses to Files

¿From a user's perspective, high performance access to files is important, whether the underlying mechanism involves parallel accesses or something else. It is clear, however, that given technological trends, extending parallelism to I/O is the only way to achieve high performance in I/O. Parallelism in accessing files has an implication on the language from which the parallel accesses will be performed. Fortran (on which HPF is based) file I/O has sequential semantics. There are several questions to be resolved for implementing parallel accesses to files from HPF. Is it necessary or important to provide an explicit mechanism to perform parallel accesses to a file using new constructs such as parallel read and write statements? If yes, how do these statements interact with traditional (sequential) I/O semantics? Some preliminary work has been done in this area [14, 21, 22].

Given that parallel access to files is needed, is it necessary to define new types of files for this purpose? Some relevant proposals are included in the HPF *Journal of Development* [54] and in [22, 28].

2.3.2 Striping/Distributing Files over Disks

Given that parallelism is crucial to achieving high performance in I/O, files are and will be striped over disks. Several important questions to address must be addressed. What controls does a user have in specifying such distributions? Does the user see the individual stripes (or portions) of files from within an HPF program? Are Fortran/HPF semantics preserved in such a file? These questions are obviously related. Most parallel machines implement some sort of striped file system, and many allow control on stripe size, data distribution, number of disks, etc.

To take advantage of the above features, it has been argued that future HPF extensions should allow an option of breaking away from traditional sequential Fortran I/O semantics, at least for some special types of file. This argument is similar to the one on which distribution of data over processor memories is based in the HPF model. Examples of these files include scratch files that may exist during the execution of a program to store temporary data. Furthermore, in many applications, it is desired to write data in one distribution and read it in another (potentially by a different number of processors). Following the current language semantics, accesses in most cases will have to be sequentialized.

Various proposals for distributing files over disks are summarized in the HPF *Journal of Development* [54]. Other work in striping files and distributing over disks is discussed in [15, 21, 27, 22, 57]

2.3.3 Out-of-Core and Persistent Arrays

Many potential applications of HPF operate on large quantities of data. Primary data structures for these applications reside on disks. These data structures are termed *out-of-core*. Specifying out-of-core data structures within HPF means that application developers are not restricted to problem sizes that fit within available memory. Mechanisms for providing this feature includes extending the directives in HPF to declare out-of-core arrays and their distributions on disks. Initial work in this area has been described in [12, 13, 89, 90].

Persistent arrays are those that persist beyond a program's execution – for example, data produced by one program and later needed by another. Metadata associated with these arrays may describe distribution and access mechanisms. Persistent arrays are perhaps more naturally considered in the Fortran 90 standards process rather than as an HPF extension.

2.3.4 Checkpointing/Restart

The ability to checkpoint an HPF program's state and then restart the program is especially critical in applications that execute for a very long time, as these applications are particularly likely to suffer from machine or software failures. As checkpointing time should be minimized, efficient parallel I/O for checkpointing is required. Another important requirement is the ability to restart a computation on a different number of processors than the number of processors on which the program was executing when the checkpoint was taken. These requirements have clear implications for the language, compiler, and runtime system. If checkpointing information is dependent on data distribution or number of processors, then restarting on a different number of processors may be very difficult, if not impossible.

Language features or directives can help compilers determine good places in programs to perform checkpoints. Just as users can use their knowledge of the application domain to provide information on data mapping and interactions, users can also provide information on which data to checkpoint and where. This information reduce both the burden on the compiler and the amount of information to be saved in a checkpoint.

This feature is also a good candidate to be included in the base Fortran 95 language because checkpoint and restart may be needed independent of any parallel execution. However, for issues such as an ability to restart on a different number of processors, HPF seems a more appropriate place.

2.3.5 Asynchronous I/O and Prefetching

Because I/O is slow compared with computation and even communication, it is attractive to overlap computation with I/O as much as possible. Such overlapping requires the ability to perform asynchronous I/O and user-level prefetching of data. There are three possible approaches to providing prefetching and asynchronous I/O capabilities: runtime libraries, language extensions, and directives.

Runtime libraries can be used for asynchronous I/O, much as libraries are used for irregular computations and reductions. This approach requires the fewest language extensions. However, since for asynchronous I/O control returns to the program while I/O is going on, the mechanism for signaling completion and synchronization is not clear.

Using directives for asynchronous I/O is another option. Again, issues of signaling and data pollution need to be resolved. The following example illustrates the use of directives. The basic idea is for the user to specify a region in which the user guarantees not to access the buffers involved in the I/O. Thus, asynchronous I/O may be performed while computation within the region is going on. If the I/O does not finish within this region, then a mechanism is required to block further computation until the I/O completes.

```
        READ (7,30) A(:,:), Q,Z
!HPF2$  I PROMISE NOT TO TOUCH THE I/O VARIABLES
C Specifying that the variables involved in the preceding read operation
C will not be accessed between this directive and its corresponding
C end of region directive
        B=TRANSPOSE (MATMUL(C,D))
        CALL TEDIOUS_CRUNCHING
!HPF2$  NOT!
C A directive to specify end of user guarantee.
        Z= SUM (A) *Z
```

Many open issues must be addressed if this mechanism is to work. These include: how to handle error conditions, how to specify completion of I/O, and how to inquire about completion.

2.4 Communication Optimizations

All interprocessor communication in HPF is implicit. For example, the shift of data that occurs in a finite difference stencil is expressed in HPF in any of the following ways:

- With a forall statement referencing A(I-1), A(I), and A(I+1);
- With array sections such as A(2:N-1), A(1:N-2), and A(3:N);
- With the EOSHIFT intrinsic function.

Shift communication may also be subtly expressed as array assignment in which the left hand side is not aligned with the right hand side:

```
        REAL A(101), B(100)
!HPF$   ALIGN B(I) WITH A(I)
        A(2:101) = B
```

Finally, the realignment of an array may imply a shift communication:

```
!HPF$   ALIGN B(I) WITH A(I)
        ...
!HPF$   REALIGN B(I) WITH A(I+1)
```

Some other regular communication patterns are stated idiomatically in HPF. For example

```
        A = TRANSPOSE(A)
```

is a transpose, as is

```
!HPF$   ALIGN A(I,J) WITH B(I,J)
        ...
!HPF$   REALIGN A(I,J) WITH B(J,I)
```

and

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

Finally, irregular communication is stated in HPF through a use of vector-valued subscript. HPF allows for several forms:

```
        A = B(V)
        FORALL(I=1:N) A(I) = A(I) + B(V(I))
        B(V) = A
        B = SUM_SCATTER(A, B, V)
```

Not every use of a vector subscript, however, is a communication. An important idiom expresses table lookup without communication:

```
        REAL PARAMS(200)
        REAL A(NUMBER_OF_PROCESSORS())
        INTEGER INDEX(NUMBER_OF_PROCESSORS())
!HPF$   DISTRIBUTE (BLOCK) ::  A
!HPF$   ALIGN PARAMS(*) WITH A(*)
!HPF$   ALIGN WITH A :: INDEX
        FORALL (P = 1:NUMBER_OF_PROCESSORS())  A(P) = PARAMS(INDEX(P))
```

In all cases, the HPF compiler is responsible for the efficient implementation of the required communication on the underlying runtime software and hardware.

The proposed HPF extensions allow the user to provide additional information, without which certain communication optimizations will not be applicable by the compiler. In most cases, the user asserts facts about a program that cannot always be deduced automatically. As with the INDEPENDENT directive, the compiler must use this additional information appropriately.

2.4.1 Asserting the Reuse of a Communication Pattern

Suppose the following code appears.

```
        INTEGER V(N)
        REAL A(N), B(N)
!HPF$   ALIGN (I) WITH A(I) :: B, V
!HPF$   DISTRIBUTE A(BLOCK)
        DO K = 1, M
          FORALL(I=1:N) A(I) = A(I) + B(V(I)).
          <rest of loop>
        ENDDO
```

The implementation of the vector gather implied by B(V(I)) is quite involved. The best-known technique for this is the inspector/executor paradigm developed by Saltz and others [83, 60]. It exploits the fact that the communication can be divided into a phase that depends on the addressing pattern (V above) but is independent of the communicated data (B above), and a phase that requires this data. To be specific, let us assume that the addition to A(I) is to be performed by the processor owning A(I). On a message-passing system, each processor must find the destination processors for each element of B that it owns, assemble messages to be sent to those processors, call the message passing system (which imposes significant additional startup burdens), and then have the data sent. If V(I) is aligned with A(I), then knowledge of the processors to which elements of B are to be sent cannot be obtained with local information, but rather requires a preliminary communication step in which gathering processors send requests for elements to their owners. Finally, when elements of B arrive at a processor that needs them, they must be buffered into a compiler-allocated data structure; the global indices V must be converted into local indices, used to find the arriving data, so as to avoid having local buffers of size SIZE(B).

Note that if and only if the value of the vector subscript V is the same at each iteration of the K loop, almost all of this work can be "hoisted" out of that loop and amortized over the M uses of the communication pattern. A good optimizing HPF compiler will do so.

Unfortunately, the compiler cannot always see enough context, because of control dependences and separate compilation, to know whether a vector subscript is used many times. What is proposed is a directive to assert the reuse of this and possibly other communication patterns, and to specify the extent of the reuse. The design of such a directive should consider past experience with other methods:

- **Compiler analysis.** Invariant schedules can frequently be detected in practice, given extensive compiler infrastructure [25, 49]. Interprocedural analysis may be necessary [1].
- **Timestamping.** Extensive analysis can be avoided by maintaining time stamps for the variables used to compute a schedule. Each modification of a variable is paired with an update to the variable's timestamp. A schedule may be reused if all inputs have the same timestamps at the point of reuse as at the point of original computation [75].
- **User specification.** The PARTI, CMSSL, and CHAOS libraries have been used to specify schedule reuse by hand at a rather low level. Constructs to be added to HPF should be designed at a much higher level.

2.4.2 Split-Phase Dereference of Distributed Objects

In general, compilers should try to initiate communication as early as possible, so that communication latency can be overlapped with computation not requiring the data being communicated. The proposal here is to allow the user to specify to the compiler that certain communications should be initiated. Consider, for example, an asynchronous assignment statement, or an asynchronous `REALIGN` directive, that initiates communication with an automatic stall when the realigned data is first referenced.

2.4.3 Locality Assertions

A proposal in the HPF *Journal of Development* [54] discusses `EXECUTE_ON_HOME` directives together with locality assertion. These two mechanisms are intertwined to the extent that we will not attempt to separate them here. The `ON_HOME` clause allows the user to assign a loop iteration to a single processor, indirectly specified as the owner of an array element. A `LOCAL_ACCESS` <variable-list> clause allows the user to assert that all references to the named variables in the loop are to elements stored on the executing processor.

We should also consider assignment of iterations to processor sections or fully general processor subsets, as the following shows. Consider an implementation of Quicksort as a recursive HPF procedure. In the recursive subroutine `QUICK`, one finds the following.

```
!HPF$  INDEPENDENT
       DO I = 1, 2
           IF (I == 1) CALL QUICK(LEFT_VALS)
           IF (I == 2) CALL QUICK(RIGHT_VALS)
       ENDDO
```

(Note the proposal in Section 2.2.5 for a "parallel sections" construct that would replace this clumsy expression of tree-based parallelism.) Here we have placed the elements of the input vector that are less than some specific element into `LEFT_VALS` and the remainder into `RIGHT_VALS`, possibly using the `PACK` intrinsic. HPF-2 may allow us to specify a set of processors to be used in each of the calls. (See, e.g., the proposed extension allowing sections of processors arrangements in `DISTRIBUTE`.) We could also arrange to map the data to the processors, as follows:

```
!HPF2$ DISTRIBUTE  LEFT_VALS ONTO PROCS(LOPROC(1) : HIPROC(1))
!HPF2$ DISTRIBUTE  RIGHT_VALS ONTO PROCS(LOPROC(2) : HIPROC(2))
!HPF2$ INDEPENDENT, EXECUTE (I) ON PROCS(LOPROC(I) : HIPROC(I))
       DO I = 1, 2
```

```
          IF (I == 1) CALL QUICK(LEFT_VALS)
  IF (I == 2) CALL QUICK(RIGHT_VALS)
       ENDDO
```

Finally, we would like to assert that the data required by the recursive calls is already in the right place.

```
!HPF2$ DISTRIBUTE  LEFT_VALS ONTO PROCS(LOPROC(1) : HIPROC(1))
!HPF2$ DISTRIBUTE  RIGHT_VALS ONTO PROCS(LOPROC(2) : HIPROC(2))
!HPF2$ INDEPENDENT, EXECUTE (I) ON PROCS(LOPROC(I) : HIPROC(I)), &
           LOCAL_ACCESS LEFT_VALS, RIGHT_VALS
       DO I = 1, 2
          IF (I == 1) CALL QUICK(LEFT_VALS)
          IF (I == 2) CALL QUICK(RIGHT_VALS)
       ENDDO
```

For a second example, consider the following.

```
!HPF$ PROCESSORS PROC(P)
!HPF$ DISTRIBUTE (BLOCK) ONTO PROC :: x
!HPF$ DISTRIBUTE (CYCLIC(K)) ONTO PROC :: y
!HPF2$INDEPENDENT, NEW (i), EXECUTE(k) ON PROC(k),LOCAL_ACCESSX,Y
      DO k = 0,P - 1
         DO i = 0,N/P
           y((i/K) * P * K + k * K + mod(i,K) + 1) = x((N/P) * k + i + 1)
         END DO
      END DO
```

2.4.4 Distribution Views

In programming algorithms such as the FFT in HPF, it is desirable to be able to take two different global views of the same distributed data. Suppose that the current distribution of array x is d_1. Then viewing the distribution of x as d_2 means that

1. The local view of the array on each processor is unchanged, that is, the value of i-th local element on processor p after the change of view remains the same as the value of the i-th local element on processor p under distribution d_1.

2. In the global address space, the distribution of x is considered to be d_2 , that is, the local-to-global and global-to-local indexing functions after the change of view correspond to those for d_2.

Hence, viewing the distribution of the array differently permutes the array in the global address space while retaining the same order of data elements in the local array of each processor. For example, suppose the distribution of an array of size 16 is *block* before changing its view to a *cyclic* distribution. Then, the following figure illustrates the effect of the change of view:

	P_0				P_1				P_2				P_3			
global index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
local index	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
value	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p

Viewas *cyclic*

⇓

	P_0				P_1				P_2				P_3			
global index	0	4	8	12	1	5	9	13	2	6	10	14	3	7	11	15
local index	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
value	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p

An implicit assumption of the change-of-view operation is that the number of elements on each processor is the same under both the d_1 and d_2 distribution.

The `VIEWAS` construct could have the following syntax:

`!HPF2$ VIEWAS` <*target-distribution*> :: <*array-name*>

The semantics of the `VIEWAS` construct is that the distribution of the array <*array-name*> is henceforth treated as <*target-distribution*>. No communication is required for `VIEWAS`. On a single processor, `VIEWAS` has no effect.

The use of the `VIEWAS` construct and the `LOCAL` assertion is illustrated using the example of a radix-2 Stockham FFT in Section 4.17.

2.5 Language Processor Environment

Too often, application designers find that a single language environment cannot meet their needs for high performance. Some programs have combined, for example,

- C++ (for high-level modularity and coarse task coordination) with
- Fortran (for computational kernels) and
- AVS (for end-stage data visualization).

For these high-performance applications, HPF-2 needs to define a more complete language processor environment than did HPF-1. Specifically, HPF-2 should include at least advice to implementors (and, perhaps, to the Fortran standards committee) about interoperability with extrinsic programming languages and generic third-party tools.

2.5.1 Language Interoperability

Language interoperability is important for HPF-2 to avoid "language creep" and maintain high performance. In many cases (task parallelism, highly adaptive and linked data structures, highly irregular workload coordination), a combination of a C++ layer with Kernel HPF might be a more elegant, high-performance solution than would an extended version of HPF-2. HPF-2 should explore possible standards and mechanisms for

- calling distributed extrinsic languages (parallel C++, for example) from HPF and `HPF_LOCAL`, and
- helping HPF and `HPF_LOCAL` to be called from these distributed extrinsics.

Whatever data access policies and public data structures this will require may well take a nested form, considering in turn

- basic policy and descriptor requirements for the Fortran 90 (or Kernel HPF) basis,
- additional complications introduced by HPF directives and mappings, and
- the inevitable additional "dark corners" of data descriptors and data access policies that distinguish individual vendors.

2.5.2 Tool Support

Giving HPF-2 the ability to interact with standard third-party tools (including debuggers, profilers, and visualizers) is strongly coupled to language interoperability. Many of the potential concerns (standardization of data access policies and descriptors) are the same.

The most significant additional requirement might be a static dump format that would allow third-party tools to reconstruct source-form correspondence. Third-party tool support will be especially important to small HPF vendors who cannot afford to maintain a dedicated staff of tool developers.

Section 3

Justification for Kernel HPF

HPF contains many directives that offer ease of use, but the excessive overhead associated with these directives can severely compromise performance on one or more distributed-memory parallel computers. Kernel HPF is a high-performance subset of HPF. The features chosen for the Kernel HPF subset have been selected for the following reasons.

- They allow high performance across all platforms.
- On no platform are there performance "surprises." Few, if any, combinations of Kernel HPF features unexpectedly lead the user into low-performance circumstances although it is still possible to write low-performance codes.
- They are easily understood.
- They are commonly used.
- They are valuable for all platforms.

Kernel HPF limits the directives from HPF to a core subset that can be shown to realize high performance across a broad spectrum of machines.

A short summary of the proposed Kernel HPF features follows; a more complete proposal has been developed by Meltzer.

- `HPF_Kernel` **Extrinsic** An extrinsic is provided to identify the subset to a compiler.
- `DYNAMIC Distributions` All `DYNAMIC` distributions are disallowed in Kernel HPF.
- `DISTRIBUTE` The `DISTRIBUTE` directive remains mostly intact; only `BLOCK(N)` has been removed.
- **On-Processor Dimensions** The "*" syntax is included in the `DISTRIBUTE` directive *dist-format-clause* to indicate on-processor dimensions.
- `PROCESSORS` The `PROCESSORS` directive is included without change.
- `ALIGN` The `ALIGN` directive is restricted to alignments that are direct; there may be no offsets or strides within an `ALIGN`. Replication and collapsing of dimensions are also disallowed.
- `TEMPLATE` The template in a `TEMPLATE` directive must have the same rank and extents as the arrays aligned to it.

- FORALL The FORALL statement and construct are included without change.
- PURE **Functions and Subroutines** PURE functions and subroutines are included without change.
- INDEPENDENT The INDEPENDENT directive is included without change.
- **Extrinsics** All extrinsics are included without change.
- **Intrinsics** All intrinsics are included without change.
- INHERIT The INHERIT directive is not included.
- **Subroutine Interfaces** Explicit interfaces are required for any subroutine in which data is remapped.
- **Sequence and Storage Association** There is no storage or sequence association.

Section 4

Motivating Applications

The efficient solution of many scientific problems on parallel processors requires language and compiler support beyond the capabilities of High Performance Fortran. Several members of the HPF Forum have contributed applications to illustrate these requirements. For ease of study, the selected codes that are small but include parallel idioms important to full-scale applications. The developing collection of source code and documentation can be examined at URL

`ftp://hpsl.cs.umd.edu/pub/hpf_bench/index.html`

4.1 Barnes-Hut

This code and its discussion were contributed by Robert Schreiber of RIACS. The code itself is copyright 1994, Robert Schreiber; non-profit and benchmarking uses are permitted.

4.1.1 Application Description

The application is simulation of the motion of N massive bodies under Newtonian gravity. This is a 2-D code. Extension to 3-D is simple.

The ideas are as described in papers by Warren and Salmon, Bhatt and Liu, and Singh.

The underlying problem

With N bodies there are N^2 interactions. This is prohibitive to compute when N is large. To make it faster, bodies are grouped into a hierarchy of clusters that each occupy some box in space. If an body is far from a box, then the force on the body due to all the bodies in the cluster is approximated as the force due to one body with the combined mass of the cluster, at its center-of-mass. The clusters are the bodies in a quadtree hierarchy of boxes. The root is a box containing all bodies. Its children are four boxes of half its length and width. The dissection of space proceeds recursively until clusters having just one body are reached. This quadtree is called the BH tree. It is a function of the positions of the bodies.

Favored parallel algorithms

At each time step, the accelerations of the bodies are computed, and their velocities and positions are advanced by a Runge-Kutta scheme with some adaptive time stepping. The accelerations are computed by an independent loop over the bodies, as the force on an body is a function of its parameters and the BH tree. Before these can be obtained, however, the BH tree is constructed.

The process of constructing the BH tree is written as a recursive F90 procedure in an obvious way. To control the work, at each recursive call the procedure is passed a section of the body array containing the bodies in its box and no others. This recursion can be carried out in parallel. There is nonnegligible data parallelism at each invocation, too.

The force calculation is an independent loop with a call to a recursive depth first traversal of the BH tree with pruning when the body is far enough from a box.

Detailed algorithm

Pseudo-code:

```
type (bh_cell)
   real data
   type (bh_ptr), dimension(4) :: pkids
end type
type (bh_ptr)
   type (bh_cell), pointer :: node
end type

type (particle), dimension(nparts) :: parts
type (bh_cell), pointer :: root, make_bh_tree

do t = 0, t_final, delta_t
  root = make_bh_tree(1, nparts)
  forall (i = 1 : nparts)
              force(i) = force_on_part(p(i), root)
      parts(i)%position = parts(i)%position + &
                          delta_t * parts(i)%velocity
      parts(i)%velocity = parts(i)%velocity + &
                          delta_t * (force(i) / parts(i)%mass)
  endforall
  call deallocate_bh_tree(root)
enddo

pure, recursive, type (bh_cell) FUNCTION make_bh_tree(lop, hip)
integer lop, hip, lo(4), hi(4)
pointer make_bh_tree

allocate(make_bh_tree)
if (lop == hip) then
  make_bh_tree% data = <mass, center of mass, etc>
else
  compute populations of 4 half-boxes,  set lo, hi;
          rearrange particles in parts(lop:hip) to move data in each
           subbox to contiguous piece of parts;
  !HPF$ independent
  do i = 1, 4
            make_bh_tree%pkids(i)%node => make_bh_tree(lo(i), hi(i))
```

```
     enddo
    endif
end FUNCTION

pure, recursive, type (point) FUNCTION force_on_part(p, tree)
type (particle) :: p
    type (bh_cell), pointer :: tree

if ( (tree%pop == 1) .or. &
       norm( p%position - tree%center )  >  tree%diameter ) then
!!! far away; treat as a lumped mass
  force_on_part = G*m1*m2 / r**2   !!! freshman physics
else
!!! must look at the subboxes.  Note cute elemental invocation in p-code :-)
  force_on_part = sum( force_on_part(p, tree%pkids(:)%node) )
endif
end FUNCTION
```

Data structures and layout

Two derived types, one for BH cells and one for bodies are employed. To enable data-parallel operation on sets of bodies, there is a single array of this type, in which all bodies reside. On the other hand, because the BH tree is created by recursive calls and because its topology is not known a priori, and because there are no data parallel operations performed on it (except to deallocate it at each timestep!) it consists of freestanding, individually allocated scalars of type BH-cell.

The layout, or mapping, of both these data structures is irregular and dynamic.

4.1.2 Workarounds for HPF-1

For HPF-1, one might use a naive, block mapping of the body array, and one might embed the BH-tree in a complete quadtree of some maximum depth, in which case a canonical assignment to the elements of an array may be used. This array may likewise be block mapped. There is some hope of maintaining locality in the body array by permuting its entries at each timestep. The parallelism of the force calculation is simple to express as an HPF independent loop, and the owner-computes rule suffices in this case for the work assignment. The code will run very slowly, however, because of irregular and data-dependent access to the distributed BH tree.

Similarly, one can invoke a recursive procedure in the body of an independent DO loop in HPF1. The difficulty in constructing the BH tree this way is to assign the work to subsets of processors and to map the array of bodies so that the processors assigned to a subtree of the BH tree (i.e. to a box in space) will own most of the bodies that lie in this space.

4.1.3 Requirements for HPF-2

Desired extensions

The source code, included on the FTP site, demonstrates the extensions needed. They are indicated with comments containing "`HPF2`". The proposed capabilities are:

- mapping scalars to single processors (for mapping the BH tree) (see Sections 2.1.4 and 2.1.2)

- partial replication in user defined, irregular mapping (2.1.6)
- distribution functions and irregular mapping of arrays (2.1.1)
- assignment of iterations of independent loops to subsets of processors (generalization of Section 2.2.1

Example pseudo-code

The use of these extensions is shown as comments in the Fortran 90 code included on the FTP site.

Context of extensions

There are numerous technical questions concerning the implementation of these extensions, and I do not believe there is consensus on their desirability, necessity, and implementability.

The most pressing concerns the process of dereferencing a pointer to a dynamically mapped target. HPF1 makes it illegal to do so without reassignment of the pointer after the remapping of the target. This would make partial replication of the BH tree less simple for the user, who would need to explicitly construct each local copy of the partial tree and relink the pointers.

4.2 ASA - Accessible Surface Area calculation

This section was provided by Shamik Sharma of the University of Maryland. The ASA code was provided by Dr. Lee through Edward Suh of NIH.

4.2.1 Application Description

The underlying problem

The *protein folding problem* is one of the most important in molecular biology. Proteins are biological macromolecules which consist of long chains of amino acids. Two key issues determine the properties of a protein (1) its constituent amino acid sequence and (2) its 3-d macromolecular structure.

Determining the macromolecular structure of a protein through computer simulation is a very computation-intensive process. Even for a small protein there may be a very large number of possible structures - and the characteristics of each structure must be examined through simulation. A critical part of the computation is the calculation of the *potential energy* of a protein structure. This function must be recomputed for every single candidate structure, so that the one with minimum potential energy can be found. The potential energy of a structure depends on a number of terms; one of the key components involves calculating the **accessible surface area** (ASA) of the molecule.

The ASA of a protein molecule is defined as the loci of the center of a spherical probe (i.e the solvent molecule) as the latter is rolled on the surface of the protein maintaining direct contact. It directly measures the degree of exposure of the protein molecule to the surrounding solvent.

Favored parallel algorithms

There are many different algorithms that have been reported for ASA calculation ([87, 79, 63]). All three algorithms cited here are parallelizable. This benchmark uses the Lee and Richards algorithm.

In the Lee and Richards algorithm, the protein molecule is modelled as a set of interlocking spheres, one for each atom. The radius of each sphere I is the sum of the Van der Waals radius of the atom i and the radius of the probe. Each atom's accessible area is computed individually and then these are accumulated to obtain the net ASA. To determine each atom's ASA, the atom is sectioned along the Z-axis at a predetermined spacing ΔZ. The intersection of each sphere with a Z-plane appears as a circle. Many portions of each circle is within other circles – these are eliminated, since the probe cannot reach these points. For each Z-section, the sum of the non-eliminated portion of each circle constitutes the *accessible arc* . The accessible arcs over all Z sections are accumulated to gibe the *arcsum*. The ASA of the atom i is then computed using the formula :

$$ASA_i = arcsum * (radius_i + radius_{probe}) * \Delta Z$$

Detailed algorithm

The computation consists of three major phases. The first phase reads in the coordinates of all the atoms in the macromolecule. In the second phase we divide the domain into cells (also called cubes) and assign each atom to a cell. In the third phase, each cell collects a list of neighbor atoms; these are atoms close enough to possibly have overlapping spheres during the ASA calculation. To generate the neighbor_atom_list, each cell includes all atoms in that cell, and all atoms in the 26 cells around it. In the fourth phase, each atom's ASA is computed. This phase divides the atom into many Z-sections. For each section, the accessible arc is computed by subtracting the arcs shadowed by neighboring atoms.

The code is structured as follows:

```
1. Break Domain into Cartesian cells
2. For each cell
     build a list of atoms neighboring each cell
3. For each cell
     For each atom in cell
       For each z section
         For each neighboring atom
           Compute ASA by computing the "Shadowing"
           effect of each neighbor atoms.
4. Accumulate  ASAs of all atoms
```

In pseudo-code the steps 2-4 look like this :

```
do j = 1 to num_cells
 do k = 1 to 27
  nc = neighbor(j,k)              ! neighbor # k
  do l = 1, num_atoms_in_cell(k)
    neighblist(j,nbsiz(j)) = atom(k,l)
    nbsiz(j) = nbsiz(j) + 1
  enddo
 enddo
enddo

do 40  j   = 1, num_cells
 do 30 l  = 1,  num_atoms_in_cell(j)
```

```
         atoma = atom(j,l)
 clear_z_sections(atoma)
         do 20 dz = 1, num_z_sections
          do  10 p = 1, nbsiz(j)
           atomb = atom(j,p)
           if ( not_neighbors(atoma, atomb)) goto 10
           if ( not_intersect(atoma, atomb)) goto 10
           shadow_overlapping_regions(atoma, atomb);
           if ( completely_overlapped(atoma) goto 30
10        continue
20       continue
30      continue
40     continue

       asa_total = 0
       do k = 1, num_atoms
          asa_total += asa(k)
       enddo
```

A break up of the time spent in each section (with data set 4mbn2 on Sun4):

1. Reading Data : 2.40 seconds
2. Building Cells : 0.04 seconds
3. Building Neigh Atom List : 0.90 seconds
4. Finding ASA of atoms : 6.70 seconds
5. Accumulations, misc. : 1.20 seconds
6. Total : 11.30 seconds

Data structures and layout

Each atom is associated with 4 doubles containing its X,Y,Z coordinates and its radius. A 5 th field of size integer, stores the cell, in which the atom is located.

Another array of the size the number of atoms stores the accessible surface area of each atom. This contains the result of the calculation for each atom.

Each cell is associated with two arrays. One array "num_atoms_in_cell" is used to store the number of atoms in that cell, while another 2-D array stores the indices of the atoms in that cell.

The largest array is the array "neighblist", which stores for each cell, a list of atoms that are in neighboring cells. This list is used to find which atoms interact with the atoms in a given cell.

A fourth set of arrays stores the arcs of each atom which have been already shadowed due to proximity with other atoms. These arrays are used inside the ASA calculation loop and are used for each atom's calculation. There is an output dependency in the use of these arrays. These arrays maintain a list of arcs that have been "shadowed" for each atom.

4.2.2 Workarounds for HPF-1

This code has several problems that make it difficult to parallelize using HPF-1. There is a reduction inside the ASA calculation loop. There are many if statements inside the loop that cause the control flow to jump out to different levels of the nested loop. The use of arrays to record the shadowed Z-sections also creates output dependences that must be removed by privatization.

4.2.3 Requirements for HPF-2

Desired extensions

- Support for reductions (see Section 2.2.2) — in this case, the reduction is simple to spot, but the compiler must know that it is considered safe to reassociate the floating-point summation of the ASA
- Support for irregular distribution of cells (2.1.1)
- Support for complicated control flow
- Ability to automatically privatize arrays that have output dependencies

The last two bullets are requirements for the compilers, not the HPF language itself.

Example pseudo-code

The computation could be expressed in an extended HPF without too many changes to the above pseudo-code. Irregular data layouts would need to be specified. The compiler must be told that floating point addition can be treated as commutative and associative for the final summation loop, so that the compiler can generate a parallel sum reduction.

Context of extensions

Scalar privatization is a common technique; the array privatization needed to remove the output dependences on the Z-sections is more complicated but still feasible. Irregular distributions and reductions have been implemented in Fortran D and Fortran 90D compilers by Ponnusamy and Hanxleden [74, 50].

4.3 Molecular Dynamics (MolDyn)

This writeup was contributed by Shamik Sharma of the University of Maryland.

4.3.1 Application Description

The underlying problem

MolDyn calculates the forces acting on each molecule in a certain domain. The domain is a cuboidal region in space and initially molecules are uniformally distributed over this domain. The initial velocities of the molecules is assigned using a maxwellian distribution. The simulation is then carried out for a number of time-steps at the end of which their final coordinates and final energies are calculated. The motion of each molecule at each time-step is determined by (a) its velocity at the beginning of the time-step and (b) the forces acting on it due to other molecules. The second part of this problem is an N-body problem.

Favored parallel algorithms

There seem to be many different sequential approaches to this problem. They mostly deal with how to approximate the force-calculation. Since the force-calculation is a N-body computation problem, I suppose all the standard N-body approximation algorithms apply here (Barnes-Hut, Fast Multipole etc.) The method used here is called the Gerard-Rokhlin algorithm (CHARMM uses a similar algorithm). This algorithm simply assumes that molecules beyond a cut-off radius from a molecule do not affect it significantly. This works well for the non-bonded electrostatic and Van der Waal's forces, which are generally not important over a long distance (unlike, for example, gravity).

The algorithm as implemented in **MolDyn** builds an *interaction-list* of molecules, listing all the pairs of molecules that can interact with each other. This list changes slowly as molecules move and must be updated frequently.

Detailed algorithm

```
a.  Initialize variables
b.  Initialize coordinates and velocities
    of molecules based on
    some distribution.
c.  DO I = 1, N time-steps
      1. Update coordinates of molecule.
      2. On Every xth iteration of I
         ReBuild the interaction-list.
         DO J = 1, NUM_ATOMS
          DO K = J+1, NUM_ATOMS
            if ( close_enough(j,k) )
              ia[cnt] = J;
              ib[cnt] = K;
              cnt++;
            endif
          ENDDO
         ENDDO
      3. Compute the force on each molecule,
         its velocity etc.
         DO J = 1, NUM_INTERACTIONS
             force(ia(j)) += foo(x(ib(i)), x(ia(i)),
                                 vh(ib(i)), vh(ib(i)) );
             force(ib(j)) += bar(x(ib(i)), x(ia(i)),
                                 vh(ib(i)), vh(ib(i)) );

         ENDDO
    ENDDO
e.  Using final velocities, compute KE and PE of system.
```

The computationally intensive part of this application is the force-computation loop that is executed each time step. This loop (an irregular reduction loop) iterates over the entire interacting list of molecules. In each iteration, an interacting pair of molecules is taken and the force acting on each due to the other is calculated. The interaction-list stores the molecule-numbers of the interacting pairs, and it is used to index into the arrays which hold information about the molecules velocity, position and forces. The force array is used to accumulate all the forces acting on each molecule.

Besides the force-computation loop, another interesting aspect of this application is the rebuilding of the interaction list every few iterations. This rebuilding of the interaction list means that any preprocessing that may have been performed to optimize the force-computation loop may have to be repeated. The rebuilding of the interaction list is not very computationally expensive.

Data structures and layout

There are 3 data arrays, `x`, `vh` and `force` — each containing double-precision numbers and each with three times as many elements as the number of molecules in the computation. The `x` array contains the 3-D coordinates of each molecule the `vh` array contains the velocity components and `force` stores the components of the force acting on each molecule.

Besides these three data arrays, there is also an interaction list that is used to index into the data arrays in the force-computation loop.

The data arrays can be distributed by BLOCK or CYCLIC (both yield similar performance), but one can obtain significant performance improvements by distributing the data using an irregular distribution which uses spatial information (the coordinates). Grouping molecules that are spatially close together in the same processor reduces the communication requirements.

4.3.2 Workarounds for HPF-1

We are mainly concerned with the force-computation loop. Since there isn't a reduction intrinsic in HPF-1 the irregular accumulation cannot be parallelized. However one can extract significant parallelism out of the loop by creating temporary arrays to hold the results of the force-computation of each interacting pair. The computation of the forces can thus be completely parallelized — only the accumulation into the force arrays would then need to be done sequentially. Besides the fact that the accumulation is not completely parallelized, this approach also imposes a signifiacnt memory overhead, since the interaction list can be huge and one would need to create temporary force arrays (doubles) of that size.

Actually the sum-reduction can be recognized despite the indirection, but there needs to be support not only for the recognition, but for the generation.

4.3.3 Requirements for HPF-2

Desired extensions

- Irregular Mapping of Arrays (2.1.1)
- Parallel reductions (2.2.2)
  All that is really required here is the assertion that the floating-point summations can be rearranged.

Depending on the level of compiler support expected, the following additional extensions may help or hinder optimization:

- Iteration Mapping (2.2.1)
  The mapping of iterations to processors should be done in conjunction with the mapping of the irregular arrays to maximize locality.

- Communication Pattern Reuse (2.4.1)
  Acceptable levels of reuse should be achievable automatically by the compiler.

Example pseudo-code

The main loop requires minimal rewriting with support for map arrays and a Fortran-D-style reduction intrinsics.

```
DISTRIBUTE maparray force, vh, x

DO I = 1, N_STEPS
  ....
  DO J = I, NUM_INTERACTIONS
    reduce(sum, force(ia(j)),
           foo(vh(ia(i),vh(ib(i),x(ia(i)), x(ib(i))) );
    reduce(sum, force(ib(j)),
           bar(vh(ia(i),vh(ib(i),x(ia(i)), x(ib(i))) );
  ENDDO
ENDDO
```

Context of extensions

Reduction intrinsics have been proposed by the Fortran-D project. Raja Das, Reinhard von Hanxleden and Ravi Ponnusamy have implemented features similar to the DISTRIBUTE maparray in the Fortran-D and F90D compilers, and demonstrate significant improvements in codes similar to **MolDyn**.

4.4 Non-bonded Force Calculations with Cut-Off

This mini-application and and its writeup were contributed by Raja Das of the University of Maryland.

4.4.1 Application Description

The underlying problem

In any molecular dynamics computation a substantial portion of the time is spent performing the non-bonded force calculation. The non-bonded forces are those that arise due to electro-static and Vandes Waal's interaction between the atoms of the molecules. Since every atom interacts with every other atom this is a N^2 calculation, but in practice forces are calculated for atoms within certain distances (cut-off). This is still a O(N^2) calculation, but actual parameter with which N^2 is multiplied is very small ($\ll 1$). Most of the industrial strength molecuular dynamics code like CHARMM, AMBER, GROMOS etc. performs the non-bonded force calculations with a cut-off.

Favored parallel algorithms

First of all, I believe there are many different sequential approaches to this problem. They mostly deal with how to approximate the force-calculation. Since the force-calculation is a N-body computation problem, I suppose all the standard N-body approximation algorithms apply here (Barnes-Hut, Fast Multipole etc.) The algorithm used here works as follows:

1. Non-bonded list generation is done to find the atoms interacting with each other. This is done every few time-steps (between 25 to 100).
2. Since this list is not updated every timestep, during the actual force calculation there is a further check to see if the atoms have drifted outside the cut-off distance.
3. After the force calculation the atom positions are updated.

Detailed algorithm

```
a.  Initialize variables
b.  Initialize coordinates and velocities
    of molecules based on
    some distribution.
c.  Initialize Non-bonded list
d. Compute the force on each molecule,
   its velocity etc.
   ITEMP = 0
   DO I = 1, Num_Atoms
     NPR = INBLO(I) - ITEMP
     DO J = 1, NPR
       k = JNB(J+ITEMP)
       if (distance(x(k),x(i)) < cutoff) then
         force(i) += foo(x(k), x(i))
         force(k) += bar(x(k), x(i))
       endif
     ENDDO
     ITEMP = INBLO(I)
   ENDDO
```

The computationally intensive part of this application is the force-computation loop that is executed each time step. This loop (an irregular reduction loop) iterates over the entire interacting list of atoms. In each iteration, an interacting pair of atoms is taken and the force acting on each due to the other is calculated. The interaction-list stores the atom-numbers of the interacting pairs, and it is used to index into the arrays which hold information about the atom coordinates. The force array is used to accumulate all the forces acting on each atom.

Besides the force-computation loop, another interesting aspect of this application is the rebuilding of the interaction list every few iterations. This rebuilding of the interaction list means that any preprocessing that may have been performed to optimize the force-computation loop may have to be repeated. the rebuilding of the interaction list is not very computationally expensive, it is also not easily parallelizable. In this example we don not present this part of the code.

Data structures and layout

There are a number of data arrays but the important ones are the ones that store the coordinate information (x, y and z), the force information (dx, dy and dz), the interaction list (jnb) and the pointer into the interaction list (inblo). The interaction list is a sparse matrix stored in a CSR format.

The data arrays can be distributed by BLOCK or CYCLIC (both yield similar performance), but one can obtain significant performance improvements by distributing the data using an irregular distribution which uses spatial information (the coordinates). Grouping molecules that are spatially close together in the same processor reduces the communication requirements.

4.4.2 Workarounds for HPF-1

We are mainly concerned with the force-computation loop. Since there isn't a reduction intrinsic in HPF-1, the irregular accumulation cannot be parallelized. However one can extract significant parallelism out of the loop by creating temporary arrays to hold the results of the force-computation of each interacting pair. The computation of the forces can thus be completely parallelized — only the accumulation into the force arrays would then need to be done sequentially. Besides the fact that the accumulation is not completely parallelized, this approach also imposes a significant memory overhead, since the interaction list can be huge and one would need to create temporary force arrays (doubles) of that size.

Actually the sum-reduction can be recognized despite the indirection, but there needs to be support not only for the recognition, but for the generation.

Another feature of this code is the presence of an if statement inside the inner loop. During the execution of this loop on a distributed memory machine, depending on how the data is distributed coordinate data needs to be fetched and force data needs to be written out to memory locations in other processors. Because of the presence of the if statement in the inner loop the number of coordinate data read from other processors and the number of force data updated in other processors memory is different. To perform message blocking to improve performance there is a need to execute the if statement outside the loop to find the actual number of off-processor updates and in what memory locations. This if-statement can be thought of as inducing another level of indirection in the code. A program slicing technique can be utilized to generate efficient code for this.

4.4.3 Requirements for HPF-2

Desired extensions

The required extensions are the same as for **MolDyn** given above. More sophisticated compiler handling is required because of the conditional inside the loop.

Example pseudo-code

```
DISTRIBUTE maparray force, vh, x

DO I = 1, N_STEPS
  ....
  ITEMP = 0
  DO I = 1, Num_Atoms
    NPR = INBLO(I) - ITEMP
    DO J = 1, NPR
      k = JNB(J+ITEMP)
      if (distance(x(k),x(i)) < cutoff) then
        reduce(add,force(i),foo(x(k), x(i)))
        reduce(add,force(k),bar(x(k), x(i)))
      endif
    ENDDO
```

```
          ITEMP = INBLO(I)
        ENDDO
      ENDDO
```

Context of extensions

Reduction intrinsics have been proposed in the Fortran D project. The program-slicing technique to unroll multiple levels of indirection has been implemented in the Fortran D compiler.

4.5 EULER: A Multimaterial, Multidiscipline, 3–D Hydrodynamics Code

4.5.1 Application Description

The EULER benchmark is derived from the PAGOSA code, developed at Los Alamos National Laboratory by Kothe, Baumgardner, Cerutti, Daly, Holian, Kober, Mosso, Painter, Smith, and Torrey. This description and following subsections contain much material quoted or adapted from their report [61]. PAGOSA, in turn, owes its basic physical model and numerical algorithms to MESA [56].

The PAGOSA code incorporates a "new data–parallel model for three–dimensional (3–D) high–speed fluid flow and high–rate material deformation... The model... has been developed recently by a team of computational physicists, numerical analysts, and computer scientists at the Los Alamos National Laboratory (LANL) on the Connection Machine (CM) series of massively–parallel supercomputers. With its efficient material interface reconstruction algorithm and finite–difference approximations on an Eulerian mesh, PAGOSA is well–suited for modeling transient flows involving multiple immiscible fluids and/or distinct materials experiencing large distortion. The evolving suite of physical models in PAGOSA currently includes models for compressible hydrodynamics, realistic equations of state, elastic–plastic material deformation, reactive burn of energetic materials, neutron transport, and turbulence effects. The PAGOSA algorithms are clear, concise, and portable due to implementation in data parallel fashion with the Fortran 90 programming language."

As a derivative of PAGOSA, EULER is a multi–model, as well as multi–material code. However, for the purposes of the HPF II benchmarks, the multi–material aspect is of most importance. EULER has been adapted by the developers and the Los Alamos benchmark group (Olaf Lubeck et al), and is considerably reduced in scope and length from the PAGOSA production code. While the original developers consider the PAGOSA code a success on machines such as the Connection Machine, the developers and the benchmark group have analyzed performance in some detail. Their conclusion is that the CM implementation can be quite inefficient due to the low "truth ratio" of array operations implied by the actual multi–material data distribution. In other words, data arrays are allocated for all materials, even though many regions of the problem may not actually contain some of the materials. Furthermore, calculations are done for all materials, even in regions where the density of materials is zero. So, much computational power is wasted on meaningless calculation.

The underlying problem

EULER's parent PAGOSA is intended to solve detailed physical simulations involving fluid flow and material deformation, with mixed materials, highly complex 3–D geometries, and a number of options for simulation physics. As one example, the development team completed a "simulation of a problem important to the oil industry known as oil well perforation. Well holes are typically lined with steel pipe and/or concrete casing that usually must be perforated with tiny high–explosive charges ('oil well perforators') prior to pumping. Perforation allows production of oil from specific depths predetermined from logging data. The perforators are inserted into the well hole inside 'carrier tubes', and then detonated when the tube has been lowered to the prescribed depth. They are designed to make clean holes in the casing and to penetrate several inches outward into the surrounding oil–bearing strata. By modeling the perforation process, PAGOSA can be used to study perforator performance, i.e., hole size and penetration depth, as a function of perforator design and layout, tubing/casing geometry, rock formation, and other design parameters."

Favored parallel algorithms

The EULER/PAGOSA model incorporates algorithms to solve for the time evolution of each material based on the continuum mechanical conservation relations, and using a "one–fluid" approximation in which all materials move with a single velocity. The system of conservation equations is closed with constitutive relations for the material pressure and deviatoric stress. These equations are solved in Eulerian (i.e. fixed) frame, "partitioned in Cartesian geometry into fixed, logically connected, orthogonal hexahedra... The conservation equations are solved using second–order accurate finte difference approximations with a conventional Lagrangian/remap scheme."

Data structures and layout

The basic data structure of the problem is a distributed, 3–D mesh: The computational domain is partitioned in Cartesian geometry into fixed, logically–connected, orthogonal tetrahedra. Flow variables are laid out on the mesh in a standard 'staggered mesh' fashion, with each component of the velocity field residing at vertices and all other variables located at cell centers.

The multi–material implementation is handled as follows: The EULER/PAGOSA "parallel array dimensions (those that are spread across the processors) correspond to the (x,y,z) coordinates. The other dimension, needed for multiple materials, is 'serial', or an in–processor dimension that exists entirely within the memory space of each processor. Elements of all arrays with identical spatial coordinate indices therefore reside within the same physical processor. The CM Fortran compiler currently lays data out on the machine by allocating a portion of each parallel array to each processor, termed the 'subgrid' because it represents a subvolume of the mesh."

"Communication of data is almost entirely nearest–neighbor, termed 'NEWS' on the CM (for North–East–West–South), and is accomplished with the Fortran 90 CSHIFT and EOSHIFT functions. The few exceptions are the global reductions (reducing the data in an array to one value, such as a maximum) used for diagnostics and time step reduction. Communications are only nearest–neighbor because PAGOSA is based on a time–explicit finite–difference formulation using a logically–connected data structure mapped onto a regular Cartesian mesh. The explicit nature minimizes the number of needed global reductions and the regular data structure eliminates the need for non–local communications, both of which degrade performance on the CM..."

4.5.2 Workarounds for HPF-1

"... The PAGOSA algorithms are highly parallelized and well-suited for the CM, as current performance analysis indicates that ¿95computational time is spent in parallel operations on the CM, only 10-20parallel computation time, however, is currently spent performing useless work as discussed below.

Algorithms that may not parallelize easily are those that require logical branching. The Fortran 90 WHERE construct, used in conjunction with masks, however, significantly alleviates the problem, and enables parallel array operations to be conditional. Legislating that different operations be performed on different portions of an array is also allowable without breaking parallelization. This feature is used in PAGOSA to great advantage, for example, in the application of boundary conditions and in the upwind differencing of the advection calculation.

A component of the PAGOSA algorithm that did not parallelize easily is the treatment of cells containing more than one material ('mixed cells'). Despite the fact that the percentage of mixed cells at any given time is usually quite small, the current PAGOSA algorithm treats every cell as though it were mixed, with memory allocated accordingly, even though the cell may contain only one material. A large load-balance penalty is therefore incurred frequently, since for problems with many materials most processors are doing useless work a large fraction of the time. This also results in inefficient memory usage for problems with large (¿ 10) numbers of materials. It is surprising that, in spite of this load-balance problem, PAGOSA performs well on the CM relative to MESA [56], a similar 3-D flow model written for conventional vector machines. This improved performance has prompted other groups to follow suite, using the same basic data parallel implementation [72]."

It should be noted that although there exist inefficiencies in this implementation, the physicists who use PAGOSA are able to run problems on the CM which they cannot run elsewhere, due to its large amount of memory.

4.5.3 Requirements for HPF–2

Desired extensions

Currently, a definitive solution to the problem is not known to the developers and benchmark group. (At least in the context of a data–parallel programming model.) One approach might be to consider a set of sparse spatial meshes, each of which comprises only one material, and which is distributed over some subset of the processors. This would seem to require some combination of irregular mappings (Section 2.1.1) and mapping to processor subsets (2.1.4). However, some outstanding problems are: 1) Communications are no longer necessarily nearest-neighbor; 2) the spatial subgrids, which are themselves possibly sparse and irregular, might imply further "truth ratio" problems; 3) there is still a possible load-balance problem; 4) there still might be excessive memory consumption if arrays of indices are needed.

4.6 Multigrid (MG)

This section was contributed by Scott Baden at the University of California, San Diego. Sections 1-3 of this Multigrid writeup are based primarily on notes from by Jim Demmel at the University of California, Berkeley.

4.6.1 Application Description

The underlying problem

MG solves the discrete Poisson equation in the unit box, though it may be applied to the solution of more general systems of equations. William Briggs' *A Multigrid Tutorial*, SIAM Publishers, provides a good introduction to the topic.

As compared with traditional relaxation-based methods like Red/Black Successive Overrelaxation (RB-SOR), MG converges in a constant number of iterations, independent of the size of the

RHS. (RB-SOR converges in $O(N)$ iterations for an N^d mesh in d dimensions.) The price we pay is that one iteration of MG takes $\log_2 N$ parallel steps, though the parallel steps get successively smaller. In addition, we will not only need to communicate among nearest neighbors, but with neighbors 2 away, 4 away, 8 away, ... , N/2 away. This more distant communication is the reason for the quicker convergence, because information is "spread around" the mesh more quickly than is possible with nearest neighbor communication.

Favored parallel algorithm

Without any loss of generality, we consider the 2d algorithm; the ensuing discussion generalizes immediately to three or more space dimensions. We assume for convenience that $N = 2^n - 1$. We will need a sequence of meshes, each with about one fourth as many points as its predecessor. The finest mesh will be denoted u^0, with values ${u^0}_{i,j}$, and corresponds to the same mesh as for Jacobi and SOR. The next finer mesh will consist of every other point (in both directions) in $u_{0,i,j}$ and be denoted u_1, with entries $u_{1,i,j}$, $0 \leq i,j \leq 2^{n-1}$. u_2 will have entries $u_{2,i,j}$, $0 \leq i,j \leq 2^{n-2}$. This continues for u_k, k up to $n-1$. $u_{k,2i,2j}$ occupies the same mesh point as $u_{k+1,i,j}$.

The MG algorithm works by using an approximate solution for Poisson's equation from a coarse grid u_{k+1} as an initial guess for starting an iterative scheme for Poisson's equation on the finer grid u_k. The approximate solution on u_{k+1} is obtained by applying this algorithm recursively. On the coarsest grid, u_{n-1}, which has just 1 unknown, we solve exactly. It is easiest to describe the algorithm in terms of several simple building blocks.

Detailed algorithm

$J(\omega, k, d, u_k, b_k)$ - Weighted Jacobi's method: apply d iterations of a weighted Jacobi's method with weight ω on the grid with starting values u_k and with right hand side b_k. (For simplicity we omit superscripts indicating iteration number.)

```
for $m = 1 : d$
    forall $(1 \leq i,j \leq 2^{n-k} - 1)$
        $u_{k,i,j} = (1-\omega)u_{k,i,j} + \omega(u_{k,i-1,j} + u_{k,i+1,j} + u_{k,i,j-1} + u_{k,i,j+1} + b_{k,i,j})/4$
endfor
```

In order to use the solution from a coarse grid to help solve a problem on a fine grid, we need to be able to transfer a problem from the fine to coarse grid and back again. Going from fine to coarse is called *restriction*, and from coarse to fine is *interpolation*.

$R(k, u_k, u_{k+1})$ - Restrict the fine grid data in u_k to the coarse grid data u_{k+1}. Do this by taking 1/16 of the sum of the NW (northwest), NE, SW, and SE fine grid neighbors, 1/8 of the N, E, W, and S fine grid neighbors, and 1/4 of the fine grid value at the same location.

```
forall $(1 \leq i,j \leq 2^{n-k-1} - 1)$
    $u_{k+1,i,j} = (u_{k,2i-1,2j-1} + u_{k,2i+1,2j-1} + u_{k,2i-1,2j+1} + u_{k,2i+1,2j+1})/16 +$
        $(u_{k,2i,2j-1} + u_{k,2i,2j+1} + u_{k,2i-1,2j} + u_{k,2i+1,2j})/8 + u_{k,2i,2j}/4$
forall boundary points, let $u_{k+1,i,j} = u_{k,2i,2j}$
```

$I(k, u_k, u_{k+1})$ - Interpolate the coarse grid data in u_{k+1} to the fine grid data u_k. Do this by averaging the nearest neighbors in the grid.

```
forall $(1 \leq i, j \leq 2^{n-k-1} - 1)$ $u_{k,2i,2j} = u_{k+1,i,j}$
forall $(0 \leq i \leq 2^{n-k-1} - 1, 1 \leq j \leq 2^{n-k-1} - 1)$ $u_{k,2i+1,2j} = (u_{k+1,i,j} + u_{k+1,i+1,j})/2$
forall $(1 \leq i \leq 2^{n-k-1} - 1, 0 \leq j \leq 2^{n-k-1} - 1)$ $u_{k,2i,2j+1} = (u_{k+1,i,j} + u_{k+1,i,j+1})/2$
forall $(0 \leq i \leq 2^{n-k-1} - 1, 1 \leq j \leq 2^{n-k-1} - 1)$
    $u_{k,2i+1,2j+1} = (u_{k+1,i,j} + u_{k+1,i,j+1} + u_{k+1,i+1,j} + u_{k+1,i+1,j+1})/4$
```

Using these building blocks, we build the Multigrid V-cycle:

$MGV(k, u_k, b_k, \omega, d)$ - Perform multigrid V-cycle to solve Poisson's equation with right hand side b_k.

```
if $k = n - 1$                              // $u_k$ has just one unknown
then
    $u_{k,1,1} = (u_{k,0,0} + u_{k,0,2} + u_{k,2,0} + u_{k,2,2} + b_{k,1,1})/4$
else
    $J(\omega, k, d, u_k, b_k)$                  // perform weighted Jacobi
    forall $(1 \leq i, j \leq 2^{n-k} - 1)$      // compute residual
        $r_{k,i,j} = (-4u_{k,i,j} + u_{k,i-1,j} + u_{k,i+1,j} + u_{k,i,j-1} + u_{k,i,j+1} + b_{k,i,j})/4$
    $R(k, r_k, r_{k-1})$                        // restrict residual to coarser grid
    forall $(1 \leq i, j \leq 2^{n-k-1} - 1)$    // initial guess to coarse grid problem
        $u_{k+1,i,j} = 0$
    $MGV(k + 1, u_{k+1}, r_{k+1}, \omega, d)$    // solve coarse grid problem recursively
    $I(k, t_k, u_{k+1})$                        // interpolate coarse grid correction back to fine grid
    forall $(1 \leq i, j \leq 2^{n-k} - 1)$      // update fine grid solution
        $u_{k,i,j} = u_{k,i,j} + t_{k,i,j}$
    $J(\omega, k, d, u_k, b_k)$                  // perform weighted Jacobi again
endif
```

Data structures and layout

Each level of the multigrid hierarchy includes the right hand side, the computed solution, the residual, plus a few scratch arrays. Communication between levels is restricted to the next coarser level (if there is one) and the previous finer one (if there is one). No communication spans more than one level. Because the smoother does an equal amount of work at every data point, each level of the hierarchy can be BLOCK partitioned.

Multigrid will not make effective use of a parallel computer if N is "too small." Performance is ultimately limited by that at the finest level, and is likely to be more impressive for 3-d problems than 2-d ones. This happens because the meshes decrease in size as MG moves up the hierarchy to successively coarser levels; the cost of a sweep is proportional to the mesh size. Put differently, MG runs out of work when the mesh gets too coarse. The amount of effective parallelism that can be effectively utilized decreases with increasing level. In addition, the number of processors usable at each level will likely be smaller than that predicted by simple workload considerations: as subgrids get smaller, vector lengths get shorter, and surface to volume effects on communication become more severe. We will likely want to run on only a single processor after reaching a certain level. Each level will run on an appropriate processor subset whose size is architecture-dependent.

We may be able to reduce communication costs a bit by replicating the computation over each processor subset. This avoids the need for the processor subset which is computing the solution to communicate that result to the remaining processors. For example, if we are computing level k on all P processors, and we want to compute at level $k+1$ on $P/4$ processors, then we replicate level $k's$ computation on each quadrant of the machine. However, this may increase storage costs significantly, as described below.

Another issue that we need to be concerned with is storage. Naively, we could represent a d dimensional mesh hierarchy as hyperplanes of a $d+1$ dimensional array. However, this scheme can be extremely wasteful of storage: it consumes space for $Nlog_2(N)$ unknowns. In fact we need to represent only

$$\sum_{l=0}{}^{log_2(N)-1}(N/l)^d$$

unknowns. The difference is critical as N increases, especially in 3 dimensions.

Consider for example, for $N = 128$. We actually require storage for 2.4M unknowns. Assuming double precision floating point and 4 copies of each unknown, about 77 Mbytes of storage is used. By comparison, the naive scheme reserves space for 16.8M unknowns, or 540 Mbytes of storage. Thus, we waste a factor of 7 in storage using the naive scheme.

4.6.2 Workarounds for HPF-1

There are two issues: processor subsets and compact storage.

Processor Subsets

It is *possible* that the effect of processor subsets could be approximated through a combination of replicated storage and clever bookkeeping. In other words, the mesh at each level is the same size as the finest level, and a copy of the solution is simply tessellated the appropriate number of times to fill out the mesh, including boundary conditions. A `where` mask is needed to disable computation on the replicated physical boundaries. Whether this scheme is efficient remains to be seen; subscript expressions involving coarse and fine grid communication (restriction and interpolation) become more complicated.

Another approach is to manage recursion explicitly, that is, use separate versions of each subroutine, qualifying each subroutine name with a level number appended as a suffix. Then, following the *advice to implementors* described in §3.7, page 42, we use `PHYSICAL PROCESSORS` and `MACHINE LAYOUT` directives to set the number of physical processors used at each level. The scalar form of the `PROCESSORS` directive (p. 40) could be useful in handling serial computation. Of course, it is an interesting exercise whether the compiler can be clever enough to replicate the computations as described above.

Compact Storage

The CM Fortran code found in the benchmark suite illustrates a work-around: rather than define a monolithic mesh hierarchy structure we let each level define local, automatic storage for the required unknowns. The levels per se are not explicitly defined, but allocated and return incrementally according to the dynamics of the subroutine call sequence.

4.6.3 Requirements for HPF-2

Using textually expanded recursion is inconvenient. We prefer a dynamic form of the `PHYSICAL PROCESSORS` directive, that allows each subroutine invocation to run on a number of processors determined (at run time) by the size of the right hand side passed (see Section 2.1.4. Again it is an interesting question whether the compiler could be made smart enough to replicate the computations.

4.7 Binz - Vortex Dynamics

This writeup was contributed by Scott B. Baden at the University of California, San Diego. Portions were also written by Scott Kohn.

4.7.1 Application Description

The underlying problem

`Binz` employs vortex dynamics (specifically: Chorin's vortex blob method) to solve the vorticity-stream function formulation of the 2-d incompressible Euler equations. The method discretizes *vorticity* (the curl of velocity) onto marker particles, called vortices, and computes particle trajectories over a sequence of discrete timesteps. Computing a time step entails evaluating the force induced on each particle, and then "pushing" the particles according to the induced force. The force calculation typically dominates the computation time.

Favored Parallel Algorithm

`Binz` solves the N body problem using a rapid summation algorithm known as Anderson's Method of Local Corrections, in which the velocity evaluation is divided into two parts: far-field velocities and local corrections. In the far-field phase, particle velocities are projected onto a grid and Poisson's equation is solved to obtain an approximate discrete global velocity field; this computation is similar to that computed in particle-in-cell (PIC). This velocity field is locally corrected in the local corrections step, which recomputes nearby interactions exactly using the direct method. In this way, most of the far-field influences can be lumped, avoiding the cost of the fully direct ($O(N^2)$) computation.

The particles are sorted into a 2-d array of bins to avoid a costly $O(N^2)$ search for nearby particles. Due to the motion of the particles, the binning array is resorted after velocity evaluation. The direct local interactions are handled a bin at a time. A "correction radius" C is chosen to distinguish nearby particles, closer than C, from distant ones. These nearby particles, once identified at any time, are the ones that participate in the local part of the computation. Thus, all the particles influencing bin (i, j) are found in the bins whose indices differ from i and j by integers no bigger than C. The computation of the right and side is also organized around the bins.

Detailed algorithm

```
0. Set up initial vorticity distribution
1. Compute induced velocity on each particle
      U = 0
      a. Compute far field velocity field
         forall (i,j) in (2:M-1,2:M-1)
             accumulate velocities induced by particles in bin(i,j)
              onto positions corresponding to the boundary of U:
              U(1,:), U(M,:), U(:,1), U(:,M)

             accumulate discrete Laplacian of velocities induced
```

```
              by particles in bin(i,j) onto positions corresponding to
              local neighborhood of bin(i,j): (i-D:i+D , j-D:j+D)
         end for all

         Apply Poisson solver to solve for U

     b. Compute local corrections
        forall (i,j) in (2:M-1,2:M-1)
           Set up interpolation stencil with velocities corresponding to
              neighborhood of U(i,j)

           forall (k,l) in (-C:C , -C:C)
              Project and subtract direct velocities induced by
                 bin(i+j,k+l) from positions of interpolation stencil
           end for all

           Interpolate from corrected stencil onto particles in bin(i,j)


           forall (k,l) in (-C:C , -C:C)
              Accumulate direct velocities induced by bin(i+j,k+l)
                 onto particles in bin(i,j)
           end for all

        end forall

2. Advance each particle according to its induced velocity.
```

The `forall` loops iterate over the bins and are nearly independent except for the accumulations. Computations of the form "accumulate velocities induced by bin(.,.) on positions X" implies another nested `forall` loop, as we will compute a tensor product of velocities induced by a set of particles against a set of positions, and then apply reduction.

The sorting procedure is tricky, since particles owned by different processors can migrate to the same bin. This introduces possibly non-deterministic behavior which we will discuss below.

Data structures and layout

In the serial (f77) implementation provided, bins can have different lengths. Thus, the binning structure is implemented as an array of pointers In fact, because we manage our own storage, we implement the bins as an array of integers, that act as pointers into the particle data structure. In HPF we would store the particles in a single 1-d structure, and the bins would provide pointers into this structure. (Using a 3-d array, in which the third axis corresponded to particle number, would waste storage.)

Because the particles are distributed unevenly we must use `CYCLIC` or `BLOCK_CYCLIC` decompositions to balance the workloads. However, a significant reduction in overhead (time and memory) would result if we could utilize a recursive bisection algorithm to split the mesh large chunks carrying roughly equal work.

4.7.2 Workarounds for HPF-1

We are concerned with three problems: (1) accumulation of velocities onto the finite difference mesh, (2) avoiding non-deterministic behavior in sorting, (3) load balancing.

Accumulation onto the mesh entails computing the tensor product of velocities induced from a given set of positions on another given set of positions and then applying a reduction intrinsic to accumulate all velocities induced against a given position over all positions This does not pose any difficulty: it is a serial computation carried out in parallel for each bin, such that each accumulator is a local automatic array. The difficulty arises when we need to accumulate the local accumulators into the global mesh: the accumulators for different bins will overlap. Our solution is costly: we let each accumulator cover the entire domain of the mesh, generating a 3d structure which we can reduce with the summation intrinsic. To save storage we may be able to generate only one accumulator per processor, rather than one per bin; in which case we index the accumulator with the processor ID.

Sorting runs into a similar difficulty. Howeer, as it is relatively inexpensive on smaller numbers of processors it can be done serially with a `do` loop.

There is no way to handle recursive bisection in HPF-1.

4.7.3 Requirements of HPF-2

Desired extensions

- reduction intrinsics to handle list append and to handle summation and subtraction of *array sections* (see Section 2.2.2)
- `DISTRIBUTE(BLOCK_IRREGULAR)` to enable irregular blocked decomposition (2.1.1)

Example pseudo-code

```
integer BLOCKS(4,P)
DISTRIBUTE(BLOCK_IRREGULAR(BLOCKS)) bins
```

If language and compiler support for irregular partitioning of data and workload are provided, the high-level HPF code will bear a close resemblance to the Fortran 77.

Context of extensions

Mapping arrays are not appropriate for handling recursive bisection since they do not preserve the locality in the underlying blocking structure. Vienna Fortran's approach to handling irregular blocked decompositions may have difficulty in balancing some workloads since the partitionings are constrained to tensor products of 1-d dimensional irregular partitionings, and are not truly multidimensional.

Reduction intrinsics have been proposed by the Fortran-D community.

4.8 DSMC (Direct Simulation Monte Carlo) method

This section was contributed by Bongki Moon of the University of Maryland. The code is derived from one developed at NASA Langley.

4.8.1 Application Description

The underlying problem

The DSMC method for randomized simulation of individual particles is here used to model a real gas. It includes movement and collision handling of simulated particles on a spatial flow field domain overlaid by a Cartesian mesh [77, 100]. The spatial location of each particle is associated with a Cartesian mesh cell. Each mesh cell typically contains multiple particles. Physical quantities such as velocity components, rotational energy and position coordinates are associated with each particle, and modified with time as the particles are concurrently followed through representative collisions and boundary interactions in simulated physical space.

Favored parallel algorithms

DSMC requires efficient runtime support for movement of particle data across processors. The computational requirements at a processor tend to depend on the number of particles assigned there. Particle movement therefore may lead to variation in work load distribution among processors. The problem domain overlaid by a Cartesian mesh needs to be partitioned in such a way that work load is balanced and the number of particles that move across processors is minimized. It may also need to be repartitioned frequently in order to rebalance the work load during the computation. These characteristics raise an issue of *dynamic load balancing* and require

- effective domain partitioning methods, and
- an adaptive policy for domain repartitioning decisions.

Detailed algorithm

In the Fortran-77 version of **DSMC**, several arrays are used to store the physical quantities associated with each molecule. Most importantly for the purpose of load balancing, `cell(i)` stores the index of the cell containing molecule `i`. In this pseudo-code, the `REALIGN` directive represents an instruction to redistribute several data arrays of the same size, including `cell`, based on the value in `cell`.

```
c loop over cells to recompute physical position of each molecule

     do N = 1, n_cells
         do i = IC2(N)+1 to IC2(N)+IC1(N)
             j = lcr(i)
             p4(j) = foo(p4(j),p1(j))
             p5(j) = foo(p5(j),p2(j))
             cell(j) = foo(p4(j),p5(j))
         enddo
     enddo

c exchange molecules among processor based on new cell indices

CC REALIGN with celltemp(VALUE(cell(i))) :: p1(i), p2(i), p4(i), p5(i)
CC     \&                                    cell(i), lcr(i)
```

```
c loop over cells & molecules to reindex molecules to new cells

      do i = 1, n_cells
          IC1(i) = 0
      enddo

      do i = 1, n_moles
          IC1(cell(i)) = IC1(cell(i)) + 1
      enddo

      M = 1
      do i = 1, n_cells
          IC2(i) = M
          M = M + IC1(i)
          IC1(i) = 0
      enddo

      do i = 1, n_moles
          IC1(cell(i)) = IC1(cell(i)) + 1
          j = IC2(cell(i)) + IC1(cell(i))
          lcr(j) = i
      enddo
```

Data structures and layout

The following declaration pseudo-code defines the variables used above.

```
c create a Translation Table for cell-based data structures

C   TEMPLATE, dimension(MAXCEL) :: celltemp

c distributed array definitions

    real, dimension(MAXCEL) :: c1, c2
    integer, dimension(MAXCEL) :: IC1, IC2, maparray
    real, dimension(MAXMOL) :: p1, p2, p4, p5
    integer, dimension(MAXMOL) :: lcr, cell

C c1, c2 : physical quantities associated with cells (e.g. temperature)
C p1, p2 : x, y coordinates of molecule velocity
C p4, p5 : x, y coordinates of molecule position
C IC1 : number of molecules in a cell
C IC2 : global index of the starting molecule of a cell
C lcr : location where i-th molecule is stored,
C  i.e., if lcr(i) = j, then the information of i-th molecule
C  is stored in p1(j) etc.

c initial BLOCK distribution
```

```
C   DYNAMIC, distribute (BLOCK) :: celltemp
C   DYNAMIC, align with celltemp :: c1, c2, IC1, IC2, maparray
C   DYNAMIC, distribute (BLOCK) :: p1, p2, p4, p5, lcr, cell
```

4.8.2 Workarounds for HPF-1

To implement this code in HPF-1 would require static assignments of cells and molecules to processors, rather difficult given that the dependence of these on input data and changes dynamically during the execution. The main performance obstacle in HPF-1 is achieving good load balancing.

4.8.3 Requirements for HPF-2

Desired extensions

Highly efficient implementation of **DSMC** can be obtained provided support for irregular and dynamic distribution of data and iterations and for generalized reductions. Useful extensions include:

- Irregular Mapping of Arrays (2.1.1)
  For this application, the dynamic remapping is particularly important.

- Parallel reductions (2.2.2)
  The collection of molecules into lists for batched communication between processors can be represented as a parallel reduction. Compilers cannot detect that an procedure manipulating lists is really a set insertion operation that can be reassociated; a directive is required.

Depending on the level of compiler support expected, the following additional extensions may help or hinder optimiization:

- Iteration Mapping (2.2.1)
  The mapping of iterations to processors should be done in conjunction with the mapping of the irregular arrays to maximize locality.

- Communication Pattern Reuse (2.4.1)
  Acceptable levels of reuse should be achievable automatically by the compiler.

Example pseudo-code

The main loop requires minimal rewriting with support for map arrays and a Fortran-D-style reduction intrinsics.

The difficulties with this code involve dynamic remapping of cells and dynamic motion of particles between cells. To support remapping of cells, distributions based on map arrays are needed. Motion of particles between cells can be treated as a complex form of reduction (see discussion in requirements part).

Critical Section pseudo-code

The movement of molecules between cells can also be represented using critical sections. The straightforward translation will require communication on every iteration, but this may be acceptable on some machines and eliminated by a smart compiler on others.

The idea is to use `DO INDEPENDENT` to express the obvious fine-grained parallelism over particles and over cells, but to allow updates to shared data structures with critical sections. We use one lock per cell to guard the addition and deletion of particles to the per cell lists. An iteration only excludes others updating the same cell from concurrent entry into the critical section.

```
   type (particle) particles(nparticles)
   type (cell) cells(ncells)
   integer map_cells(ncells), map_particles(nparticles)

   do  t = 0, tmax, delta_t
   if (remapping_needed) then
      recompute the cell mapping, map_cells
!HPF2$ REDISTRIBUTE cells(map_cells)
      determine map_particles from cell member lists
!HPF2$ REDISTRIBUTE particles(map_particles)
   endif

!HPF$ INDEPENDENT NEW new_cell, old_cell
   do i = 1, nparticles [on owner clause here?]
      particles%params(i) = updated particle parameters
      new_cell = new owner of this particle
      old_cell = particles%owner(i)
      particles%owner(i) = new_cell
      if (old_cell .ne. new_cell) then
!HPF2$ CRITICAL (old_cell)(ncells)
        remove particle i from appropriate list on old_cell
!HPF2$ END CRITICAL
!HPF2$ CRITICAL (new_cell)(ncells)
        add particle i to appropriate list on new_cell
!HPF2$ END CRITICAL
      endif
   enddo

!HPF$ INDEPENDENT
     do cellno = 1, ncells [on owner clause here?]
        call local_collisions(cellno)
     enddo
enddo
```

Context of extensions

There is a limited implementation of user-defined reduction operations in the Syracuse Fortran90D compiler; the performance achieved is discussed in [85].

4.9 Sparse Cholesky Factorization

This section was contributed by Kalluri Eswar, C.-H. Huang and P. Sadayappan; all of Ohio State University ({eswar,chh,saday}@cis.ohio-state.edu).

4.9.1 Application Description

Sparse Cholesky factorization involves determining a lower triangular matrix L such that $LL^T = A$, where A is a given sparse symmetric positive definite matrix. This is the computationally dominant step in one common method of direct solution of a sparse linear system of equations $Ax = b$. The solution of large sparse linear systems of equations is commonly required in a number of scientific/engineering application domains such as Lattice-Gauge Theory, Computational Fluid Dynamics, Geodetic Modeling and Reservoir Simulation, Structural Mechanics and Dynamics, Electronic Device Simulation, and VLSI Circuit Simulation [62, 82].

It is often the case that the Cholesky factorization of several matrices with the same sparsity structure may be required. In such situations, the costs of any analysis of the sparsity structure to determine, for example, a reordering permutation for the matrix, or a mapping of the matrix elements to processors, may be amortized over several uses of the information computed, during the actual numerical factorizations.

The underlying problem

The sparse Cholesky factorization computation may be considered to consist of two main kinds of operations: the *normalize* operations, and the *update* operations. The normalize operations are performed on each column in the matrix, and involve scaling the off-diagonal elements in a column by the diagonal value. An update operation involves two columns of the matrix, one called the *source* and the other the *target*. Not all pairs of columns are necessarily involved in update operations. The set of update operations that need to be performed depends on the sparsity structure of the matrix [45]. An update operation involves subtracting multiples of some elements in the source column from corresponding elements in the target column.

The elements of a sparse matrix are stored using some compact storage scheme where only the nonzero elements of the matrix are actually stored. Accessing elements of the matrix then requires an extra level of indirection in the array subscripts, contributing to the irregularity of the computation. The sparsity structure of the matrix also determines a partial order in which the normalize and update operations may be carried out. Since the sparsity structure can, in general, be irregular, the computation structure is also usually irregular.

Favored parallel algorithms

Most of the parallel algorithms for sparse Cholesky factorization on distributed-memory multiprocessors have used the column level of granularity, *i.e.*, the matrix elements are distributed among the processors so that all elements of a column reside on the same processor [52]. The *fan-out* [44], the *fan-in* [7], and the *kji-agg-sup* [34, 32] algorithms fall in this class. Algorithms that operate at a finer level of granularity, namely, blocks of elements in the matrix, have been shown, under many situations, to reduce the communication overhead incurred during the parallel factorization. Two members of this class of algorithms are the *block fan-out* algorithm [81] and a parallel multifrontal algorithm presented in [48]. A discussion of the issues to be addressed in mapping the data and the computation of sparse Cholesky factorization may be found in [33].

Whatever the level of granularity used, the essential problem with expressing this computation using HPF is the presence of nested doacross loops, where the dependences are determined by the sparsity structure of the matrix. The canonical and baseline versions of the application provided use column-level algorithms.

Detailed algorithm

The factorization code consists of two parts: a sequential part and a parallel part. The sequential part reads in the structure of an unfilled symmetric sparse matrix. It then uses the quotient minimum degree (QMD) reordering algorithm [45] to determine a permutation of the matrix columns and rows that would result in low fill-in (extra nonzeros) in the Cholesky factor. The filled structure of the matrix is then determined. The supernodes [8] in the filled matrix are identified. Supernodes are essentially groups of columns in the matrix sharing a common nonzero structure. Exploitation of supernodes can help reduce the factorization time due to many factors, which are discussed in [34]. Since it is convenient for the columns in the same supernode to be consecutively numbered, a renumbering of the matrix columns and rows takes place to achieve this. This does not cause any change in the number of nonzeros. The matrix is then initialized with random nonzero values, with care being taken to ensure postive definiteness. The next step in the sequential part is to be determine the mapping of the columns of the matrix to processors. This is done by constructing a tree called the *elimination tree* [84] and using a heuristic called *recursive partitioning* [35, 36]. The mapping algorithm maps supernodes to groups of processors. The nodes in each supernode are distributed among the processors in its assigned group in a block fashion, causing the creation of *minisupernodes* on each processor. Nodes in a minisupernode will also have the property of sharing a common nonzero structure, as nodes in a supernode do. The matrix structure (using the final numbering chosen), other information useful for the factorization, and the matrix values are written out as a last step by the sequential part.

The parallel part is based on the fan-out algorithm mentioned above. The variant implemented by the code uses supernodes. The supernodal fan-out algorithm is presented in pseudo-code below.

```
Algorithm supernodal fan-out
  On each processor do
    count := number of owned columns
    for each owned leaf k do
      normalize column k
      if k is the last column in its minisupernode then
        send minisupernode containing k to processors needing it
      else
        send column k to myself
      end
      count := count − 1
    end
    while count ≠ 0 do
      receive a column k or a minisupernode ending in column k
      if a column k is received then
        for each column j after k in its minisupernode do
          update column j using column k (dense)
        end
      else
        for each owned column j updated by column k do
          for each column k' in minisupernode do
            accumulate update of column k' to column j (dense)
          end
          update column j using accumulated values (sparse)
          if column j is completely updated then
            normalize column j
            if j is the last column in its minisupernode then
              send minisupernode containing j to processors needing it
```

```
                  else
                     send column $j$ to myself
                  end
                  $count := count - 1$
                end
              end
            end
          end
        end
      end supernodal fan-out
```

The computation on each processor is driven by the arrival of a normalized column or a minisupernode that is completely normalized. For simplicity, it is assumed that each processor sends columns and minisupernodes to itself. A received column is used to update subsequent nodes in its minisupernode. A received minisupernode is used to update all the targets of the nodes in the minisupernode. In either case, columns that have been completely updated are normalized and sent to other processors that need to use them for update operations.

Data structures and layout

Some of the important data structures used by both the sequential part and the parallel part are:

`xlnz` This array is used to determine the number of nonzeros below the diagonal in each column. `xlnz(j+1)-xlnz(j)` gives this number for column `j`.

`nzsub` This array contains, for each column, the row numbers of the nonzeros below the diagonal in that column. The row numbers are kept in increasing order. If two consecutive columns share the same nonzero structure, this information will not be duplicated in this array. The length of this array may, therefore, be smaller than the number of nonzeros in the strictly lower triangular part of the matrix.

`xnzsub` This array contains, for each column, an index into the array `nzsub` where the row numbers of the nonzeros in that column begin. In conjunction with the array `xlnz`, the nonzero structure of each column in the matrix may be determined by looking at the appropriate portion of the array `nzsub`.

`mylnz` This array contains the actual matrix (and after factorization, the Cholesky factor) values. The values are arranged in sequence based on column number, and within each column ordered by row number. For the parallel version, this array is distributed among the processors, so that each processor's `mylnz` array contains only the columns that have been mapped onto that processor. The values continue to be stored in increasing order of the column number, and in increasing row number order within each column.

`owner` This array contains the identifying number of the processor to which each column is mapped.

4.9.2 Workarounds for HPF-1

No method other than using extrinsics and explicit message passing seems possible in HPF-1.

4.9.3 Requirements for HPF-2

Desired extensions

Sparse Cholesky factorization is characterized by the presence of doacross loops (see Section 2.2.3). HPF needs to provide mechanisms for expressing nested doacross loops. In particular, there is a need for a way to specify the partial order for the execution of the different iterations. There should also be a way to specify how the mapping of iterations to processors should be done. Quite sophisticated techniques are needed in the case of sparse Cholesky factorization for deciding the mapping of the computation. It is inconceivable that a general mapping strategy incorporated into the HPF system would be able to achieve performance close to that achieved by specific techniques such as recursive partitioning, used in the code described above.

Two constructs/clauses are suggested as possible extensions to HPF. The first is a when clause. This clause is to be used in conjunction with a do loop that has been designated as a doacross loop. The when clause could have the syntax

when <*logical variable*>

The semantics of the clause are that the iteration of the doacross loop may proceed with execution only when the logical variable in the when clause is true. The second clause suggested as an extension is a on clause, to be used in conjunction with loops of all kinds. This clause would appear somewhere inside a loop, and could have the following syntax

on <*integer expression*>

The semantics of the clause are that the iteration of the loop imemdiately enclosing the clause should be mapped onto the processor whose identifying number is given by the integer expression specified in the on clause (see Section 2.2.1).

Example pseudo-code

A fan-out algorithm for sparse Cholesky factorization is presented here that uses the extensions to HPF suggested above. To simplify the presentation, a nonsupernodal version is used. A supernodal version would make use of the new clauses in essentially the same manner as the nonsupernodal version.

```
Algorithm fan-out
  HPF2$ INDEPENDENT on owner(k)
  do $k = 1, n$
    if column $k$ is a leaf then
      normalize column $k$
      ready($k$) := true
    else
      ready($k$) := false
    end
  end
  HPF2$ DOACROSS when ready($k$)
  do $k = 1, n$
    HPF2$ INDEPENDENT on owner(j)
    do $j \in \{$ columns updated by column $k$ $\}$
      update column $j$ using column $k$
      if column $j$ is completely updated then
        normalize column $j$
```

```
L1: do time = 1, timesteps

 C Convection Phase:

     L2: do i = 1, NPOINTS
                x(i) = x(i) + F(y(i), y(i-1), y(i), y(i+1), z(i))
           end do
           y(1:NPOINTS) = x(1:NPOINTS)

 C Reaction Phase:

     L3: do i = 1, NPOINTS
                z(i) = Adaptive_Solver(x(i))
           end do

     end do
```

Figure 4.1: Overview - Combustion Code

```
                  ready(j) := true
               end
            end
         end
      end fan-out
```

Context of extensions

4.10 Flame Simulation

This section was contributed by Paul Havlak of the University of Maryland based on information in [69, 75].

4.10.1 Application Description

The underlying problem

This code performs a detailed time-dependent, multi-dimensional simulation of hydrocarbon flames. The calculation cycles between two distinct phases. The first phase (*convection*) calculates fluid convection over a Cartesian mesh. The second phase (*reaction*) solves the ordinary differential equations used to represent chemical reactions and energy release. As the chemical combustion proceeds at varying rates across the simulated space, so does the computational load varies greatly between mesh points. An adaptive method is required to balance the load.

Algorithm

Figure 4.1 presents a simplified one dimensional version of this code. The convection phase (loop nest L2) consists of a sweep over a structured mesh involving array elements located at nearest neighbor mesh points. The reaction phase (loop nest L3) involves only local calculations. The computational cost associated with the function *Adaptive_Solver* depends on the value of x(i).

It is clear that the cost of *Adaptive_Solver* can vary from mesh point to mesh point. The cost of *Adaptive_Solver* at a given mesh point changes slowly between iterations of the outer loop L1.

There are a number of strategies that can be used in partitioning data and work associated with this flame code. If the convection calculations comprise the bulk of the computation time, it would be reasonable to partition the mesh (arrays `x, y` and `z` in Figure 4.1) into equal sized blocks.

However, the reaction calculations (loop nest L3 in Figure 4.1) usually comprise at least half of the total computational cost. A majority of the work associated with the reaction phase of the calculation is carried out on a small fraction of the mesh points. The current approach involves maintaining a block mapping of the mesh (arrays `x, y` and `z`) during the convection phase. In order to ensure a good load balance during the reaction phase, only expensive reaction calculations are redistributed. In Figure 4.1, array element $x(i)$ must be transmitted in order to redistribute the reaction calculation for mesh point i. Once the reaction calculation is carried out, the solution $z(i)$ is returned to the processor to which it is assigned. At a given mesh point, the cost associated with a reaction calculation generally varies gradually as a problem progresses. This property provides a way to estimate reaction calculation costs in the subsequent computation step. Mesh points are sorted according to their predicted costs, and the most costly grid points are moved first to reduce the communication involved in balancing the load.

4.10.2 Workarounds for HPF-1

One could use a regular, static HPF-1 distribution of the data and not worry about the workload. The convection phase uses such a distribution anyway. The load imbalances in the reaction phase can cost some 12 percent of performance for "a moderately detailed reaction mechanism," and would get worse with more complicated chemistry.

4.10.3 Requirements for HPF-2

The optimizations employed in this implementation require the ability to redistribute workload in a dynamic and irregular manner (see Section 2.2.1). This implementation uses the CHAOS runtime support library, not at a particularly high level of language abstraction yet.

4.11 Fock Matrix Construction

This section was contributed by Ian Foster of Argonne National Laboratory. The benchmark Hartree-Fock code described in these notes was constructed by Robert Harrison of Pacific Northwest Laboratory.

4.11.1 Application Description

The Fock matrix construction problem arises in the Hartree-Fock method for *ab initio* computational chemistry (see [53] for an introduction).

The core of this problem is the quadruply-nested loop illustrated in Figure 4.2. Approximately N^4 integrals must be computed; each requires data from six elements of a density matrix, D, and contributes to six elements of a Fock matrix, F. Both D and F have size $N \times N$ and can be large (N is commonly in range 100–1000; chemists would like to solve problems 10 times larger). The cost of an integral is strongly data dependent, but may range from tens to hundreds of floating point operations.

```
do i = 1,ni
  do j = 1,nj
    do k = 1,nk
      do l = 1,nl
        I = compute_integral(i,j,k,l)
        F(i,j) = F(i,j) + I*D(k,l)
        F(k,l) = F(k,l) + I*D(i,j)
        F(i,k) = F(i,k) + I*D(j,l)
        F(i,l) = F(i,l) + I*D(k,l)
        F(j,l) = F(j,l) + I*D(i,k)
        F(j,k) = F(j,k) + I*D(i,l)
      enddo
    enddo
  enddo
enddo
```

Figure 4.2: Logic for Fock matrix construction problem.

An efficient parallel algorithm must both map integrals to processors dynamically and block integrals and communications so that fewer than $6N^4$ messages are required to communicate the D and F values.

Existing parallel implementations create one "worker" task per processor [51, 40]. Data are distributed in a blocked fashion. Each task both performs computation and generate requests for data sub-blocks. Requests for data arrive asynchronously and can be serviced either using an interrupt-driven receive, a polling operation, or a separate "data server" thread.

4.11.2 Workarounds for HPF-1

It is not clear how to achieve reasonable performance with HPF-1.

4.11.3 Requirements for HPF-2

This problem can be expressed using reductions or critical sections discussed in Section 2.2.2. However, this does not address the need to block and map integrals, which presumably either requires additional directives or must be discovered by the compiler.

4.12 FFT: Fast Fourier Transform (TASK)

This section was extracted from the Fx benchmarks document by O'Hallaron et al. One way to the Fx project home page is:
`http://www.cs.cmu.edu:8001/afs/cs.cmu.edu/project/iwarp/member/fx/public/www/fx.html`

The full Fx task-parallel benchmark suite is available at

`ftp://warp.cs.cmu.edu/usr/anon/fx-codes/tpsuite`

Figure 4.3: 1D FFT task graph for one input vector

Input and output are sequences of vectors reshaped as 2D arrays. Nodes labeled `trans` perform a transpose operation, nodes labeled `col FFTs` perform a set of 1D FFTs on the columns of its input array, and the node labeled `scale` multiplies each element of its input array by a constant. To exploit locality in the memory subsystem, the program implements each set of row-wise operations as a transpose followed by a set of column-wise operations. This specific order is an artifact of the fact that Fortran 77 stores matrices in column-major order. If the example program were written in C, each column-wise operation would be implemented as a transpose followed by a set of row-wise operations.

Mapping the 1D FFT onto a parallel system is easy in some ways and challenging in other ways. The problem is easy in the sense that the column-wise FFTs and the scaling operation are perfectly

Figure 4.4: 2D FFT task graph for one input array

4.12.2 Workarounds for HPF-1

Section 4.17 discusses other ways to implement the FFT.

4.12.3 Requirements for HPF-2

This implementation of FFT employs a task-parallel approach; see Section 2.2.4.

4.13 Narrowband tracking radar

This section was extracted from the Fx benchmarks document by O'Hallaron et al. One way to the Fx project home page is:
`http://www.cs.cmu.edu:8001/afs/cs.cmu.edu/project/iwarp/member/fx/public/www/fx.html`

The full Fx task-parallel benchmark suite is available at

`ftp://warp.cs.cmu.edu/usr/anon/fx-codes/tpsuite`

4.13.1 Application Description

The underlying problem

The narrowband tracking radar benchmark was developed by researchers at MIT Lincoln Laboratories to measure the effectiveness of various multicomputers for their radar applications [86]. It

Figure 4.5: Radar task graph for one input array

Favored parallel algorithms

The program inputs data from a single sensor along $c = 4$ independent *channels*. Every 5 milliseconds, for each channel, the program receives $d = 512$ complex vectors of length $r = 10$, one after the other in the form of an $r \times d$ complex array A (assuming the column major ordering of Fortran). At a high-level, each input array A is processed in the following way: (1) *Corner turn* the $r \times d$ input array to form a $d \times r$ array. (2) Perform r independent d-point FFTs. (3) Convert the resulting complex $d \times r$ array to a real $w \times r$ subarray, $w = 40$, by replacing each element $a + ib$ in the $w \times r$ subarray with its scaled magnitude $\sqrt{a^2 + b^2}/d$. (4) Threshold each element a_{jk} of the subarray using a cutoff that is a function of a_{jk} and the sum of the subarray elements. Elements that are above the threshold are set to unity; elements below the threshold are set to zero.

The corner turn operation is equivalent to a transpose, so it can potentially induce a complete exchange where each processor communicates with every other processor. As with the 1D FFT, the column FFTs, scaling, and thresholding operations can be naturally expressed using conventional data parallel constructs. Further, the reduction operation requires an efficient reduction mechanism. However, the most interesting computational property of the radar benchmark is the fact that the size parameters r, d, c, and w are determined by mother nature and the properties of current sensor technology. The luxury of simply increasing the data set size simply does not exist in this case. The amount of available low-level data parallelism is limited, so additional parallelism must come from higher-level task parallelism. Like the FFT examples, input data sets are independent, so both replication and clustering of the task graph are possible.

4.14 Multibaseline stereo

This section was extracted from the Fx benchmarks document by O'Hallaron et al. One way to the Fx project home page is:

`http://www.cs.cmu.edu:8001/afs/cs.cmu.edu/project/iwarp/member/fx/public/www/fx.html`

The full Fx task-parallel benchmark suite is available at

`ftp://warp.cs.cmu.edu/usr/anon/fx-codes/tpsuite`

Figure 4.6: Multibaseline stereo task graph for one input data set

4.14.1 Application Description

The underlying problem

The multibaseline stereo uses an algorithm developed at Carnegie Mellon that gives greater accuracy in depth through the use of more than two cameras [71]. It is an interesting program for studying task parallelism because it contains significant amounts of both inter-task and intra-task communication[99], and because, like the radar example, the size of the input data sets cannot be easily increased. Our implementation is adapted from a previous data-parallel implementation written in a specialized image processing language [98].

Favored parallel algorithms

Figure 4.6 shows the task graph for the stereo program. Input consists of three $m \times n$ images acquired from three horizontally aligned, equally spaced cameras. One image is the *reference image*, the other two are *match images*. For each of 16 disparities, $d = 0, \ldots, 15$, the first match image is shifted by d pixels, the second image is shifted by $2d$ pixels. A *difference image* is formed by computing the sum of squared differences between the corresponding pixels of the reference image and the shifted match images. Next, an *error image* is formed by replacing each pixel in the difference image with the sum of the pixels in a surrounding 13×13 window. A *disparity image* is then formed by finding, for each pixel, the disparity that minimizes error. Finally, the depth of each pixel is displayed as a simple function of its disparity.

The stereo program requires efficient mechanisms for broadcasting and reducing large data sets. The computation of the difference images requires simple pointwise operations on the three input images and can thus be naturally expressed with Fortran 90 array statements. The computation of the error images is somewhat more interesting, being similar to a convolution operation. The convolution can be modeled as a DOALL where the loop iterations operate on overlapping regions of the image, which means that processors must communicate before the loop iterations can begin executing. As with the FFT and radar programs, the data sets are independent, so both replication and clustering of the task graph are possible.

4.15 Airshed simulation

This section was extracted from the CMU Fx benchmarks document by O'Hallaron et al. One way to the Fx project home page is:

`http://www.cs.cmu.edu:8001/afs/cs.cmu.edu/project/iwarp/member/fx/public/www/fx.html`

Figure 4.7: Task graph for one hour of the airshed simulation

The full Fx task-parallel benchmark suite is available at

`ftp://warp.cs.cmu.edu/usr/anon/fx-codes/tpsuite`

4.15.1 Application Description

The underlying problem

The airshed simulation is significantly more complex than the previous examples from the Fx benchmarks. The multiscale airshed model captures the formation, reaction, and transport of atmospheric pollutants and related chemical species [66, 67]. It is an interesting application because it requires a dynamic task parallel model, and because different parts of the application exhibit widely varying amounts of DOALL parallelism.

The airshed application simulates the behavior of the airshed model when it is applied to s chemical species, distributed over domains containing p grid points in each of l atmospheric layers. Typical values are $s = 35$ species, $500 \leq p \leq 5000$ grid points, and $l = 5$ atmospheric layers. Because of the multiscale grid, the entire northeastern United States can be modeled with problems in this size range. A total of about 200 chemical reactions are modeled.

Algorithm

The program computes in two principle phases: (1) horizontal transport (using a finite element method with repeated application of a direct solver), followed by (2) chemistry/vertical transport (using an iterative, predictor-corrector method). Figure 4.7 depicts the task graph for one hour of simulated time. Input is an $l \times s \times p$ concentration array. Initial conditions are input from disk (`inputhour`), and in a preprocessing phase for the horizontal transport phases to follow, the finite element stiffness matrix for each layer is assembled and factored (`pretrans`). The atmospheric conditions captured by the stiffness matrix are assumed to be constant during the simulated hour, so this step is performed just once per hour. This is followed by a sequence of steps — the number of steps is one of the initial conditions — where each step consists of a horizontal transport phase, followed by a chemistry/vertical transport phase, followed by another horizontal transport phase. Each horizontal transport phase performs ls backsolves, one for each layer and species. All may be computed independently; however, for each layer l, all backsolves use the same factored matrix A_l. The chemistry/vertical transport phase performs an independent computation for each of the p grid points. Output for the hour is an updated concentration array, which is then input to the next hour.

A number of interesting issues arise when we map the airshed to a parallel system. In the other example programs we have discussed, the number of tasks is known at compile time. However, in the airshed program, the number of transport/chemistry/transport steps for each hour is not known until runtime, which implies a dynamic model of task parallelism. Also, since the output concentration array of one hour is the input to the next hour, replication of the task graph is not feasible, as it was with the previous example programs.

Another issue is that the preprocessing phase, the transport phase, and the chemistry phase have very different levels of obvious DOALL parallelism because the sizes of the different dimensions of the concentration array differ by orders of magnitude. For example, the preprocessing phase independently computes stiffness matrices for each layer; unfortunately there are only 5 layers, so the obvious DOALL approach will use 5 processors. To get better utilization, we must parallelize the computation for each layer, or we must try to employ task parallelism to pipeline the computation for each layer, or both.

The issues involved in mapping the transport phase are particularly interesting. Since there are $l = 5$ layers and $s = 35$ species, the transport phase could be easily implemented with doubly nested DOALLs that consist of 175 independent loop iterations. For moderate sized parallel systems, with say 64 processors, this approach might work well. However, for larger systems, with say 512 processors, this approach uses only a fraction of the processors. As with the preprocessing phase, we can get better utilization by either parallelizing the sparse finite element computation (a difficult task) or trying to use task parallelism to pipeline the computation.

The final issue stems from the fact that the preprocessing, horizontal transport, and chemistry/vertical transport phases each operate on different dimensions of the concentration array. To exploit locality in the memory hierarchy, an implementation will most likely insert the appropriate transpose operation before each phase. On a parallel system, this can induce a complete exchange where each processor communicates with every other processor. Again, as with the FFT and radar examples, we see the need for an efficient complete exchange mechanism.

4.16 Out-of-Core Matrix Transposition

This section was contributed by C.-H. Huang, S. D. Kaushik and P. Sadayappan; all of Ohio State University ({chh,kaushik,saday}@cis.ohio-state.edu).

4.16.1 Application Description

The underlying problem

Out-of-core matrix transposition involves performing the transposition $Y = X^T$ where X is a large two-dimensional matrix stored in external memory. The array size is larger than the total main memory available on the parallel machine. Since the entire array cannot fit into the collective processor memory, an algorithm which operates on portions of the array at a time is required. Out-of-core matrix transposition is used in several out-of-core applications such as the multi-dimensional fast Fourier Transform and fast Fourier Transforms of large one-dimensional vectors [95]. Very large FFT's are required for the search of faint radio pulsars and multi-dimensional FFT's are fundamental to a wide range of computational problems [5], such as those involved in the study of long range climate changes [23]. Work on developing a suite of I/O optimized out-of-core multi-dimensional FFT algorithms has been targeted [5]. The data sizes for these applications are expected to be in the range of hundred Gbytes to one Tbyte/application [29, 5]. Typical FFT applications involve square matrices with sizes which are perfect powers of two. Henceforth we consider the out-of-core transposition of an $N \times N$ matrix, where $N = 2^n$.

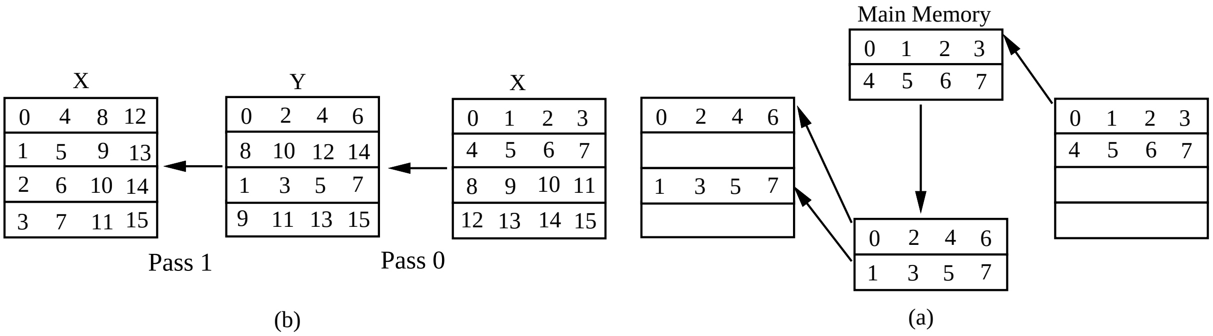

Figure 4.8: Illustration of the out-of-core matrix transposition of a 4×4 matrix.

Favored Parallel Out-of-Core Algorithm

A naive parallel out-of-core matrix transposition algorithm requires $O(N^2)$ disk accesses but performs a single pass over the input matrix, i.e., the matrix is read into main memory and stored back to external memory once. The proposed algorithm requires that at least two rows of the matrix fit in a processor's main memory and transposes an $N \times N$ matrix using $O(N log_2 N)$ disk accesses but performs $log_2 N$ passes over the input matrix. The decrease in the number of disk accesses is traded off with an increase in the data volume. If a larger amount of main memory per processor is available, the number of passes and disk accesses can be further reduced. The most general form of the algorithm requires that at least 2^k, $k > 0$ rows of the matrix fit into main memory and transposes the matrix using $N log_{2^k} N$ disk accesses and $log_{2^k} N$ passes.

The sequential version of the transposition algorithm with $k = 1$ performs n passes over the input matrix. Each pass is as follows:

1. Read input matrix into main memory two consecutive rows at a time.

2. View the two-rows as a one-dimensional array and permute the array using a stride permutation.

3. Store each row at the appropriate location in the output matrix. The rows are stored at non-consecutive locations.

4. For the next pass use the output matrix as the input matrix and vice versa.

For example, the transposition of a 4×4 matrix is shown in Fig. 4.8. The algorithm performs two passes over the matrix. The details of pass 0 are shown in Fig. 4.8(a).

In the SPMD (Single Program Multiple Data) programming model, each processor executes the same program on a local array associated with it. In an in-core program, the local array resides in the local memory of each processor. For out-of-core algorithms, the local array cannot fit entirely in main memory and is stored in external memory as a separate file. This file may be striped across multiple disks or localized to a single disk depending on the level of control the programmer has over striping and data distribution on the underlying parallel file system. This model for designing SPMD programs for out-of-core algorithms has been proposed in [91] and the local array for each processor is stored in a file referred to as the local array file (LAF). The two-dimensional array is distributed using a $(block, *)$ distribution, i.e., each processor's LAF contains a block of rows of size $\frac{N}{P}$, where P is the total number of processors. The SPMD node program for processor p performs n-passes. Each pass is as follows:

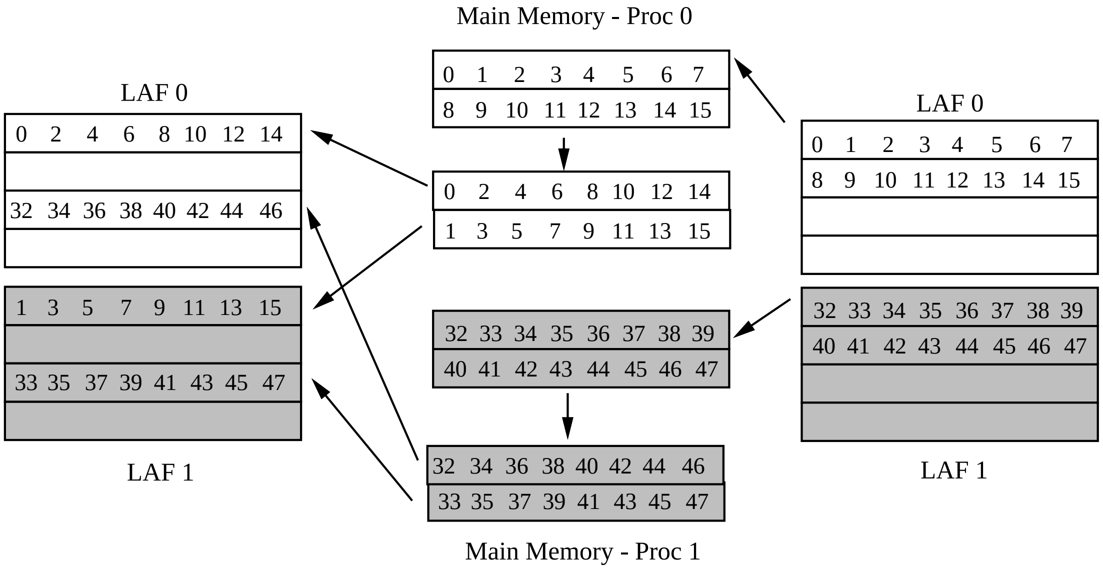

Figure 4.9: Illustration of the out-of-core matrix transposition of an 8×8 matrix on two processors.

1. Read input local array file into main memory two rows at a time.
2. View the two-rows as a one-dimensional array and permute the array according to a stride permutation.
3. Determine the locations of the two rows in the output matrix and the corresponding LAF's. Store the rows at appropriate locations in the corresponding LAF's.
4. For the next pass use the output LAF as the input LAF and vice-versa.

Note that in step 3, a processor will need to write to the LAF's of other processors. The first pass in the transposition of an 8×8 matrix on two processors is shown in Fig. 4.9.

Detailed Algorithm

Consider an $N \times N$ array A which is to be transposed and the result stored in array B. The arrays are partitioned using a $(block, *)$ distribution into P LAF's each. The LAF's corresponding to processor p contain the rows $(p * \frac{N}{P} : p * \frac{N}{P} + \frac{N}{P} - 1, 0 : N - 1)$. The node program for the out-of-core matrix transposition is shown in Fig. 4.10. In the program, $LAF\_A(p)$ is the local array file of array A corresponding to processor p and $LAF\_B(p)$ is the local array file of array B. The I/O statement read$(LAF, size, mem_addr, file_addr)$ reads a block of size $size$ starting at location $file\_addr$ in file LAF into memory starting at location mem_addr. The write statement is similar.

4.16.2 Workarounds for HPF-1

HPF-1 does not provide support for out-of-core methods. An HPF-1 implementation would have to employ low-level library and system calls, increasing the complexity and reducing the portability.

```
do $k = 0, n - 1$
   do $i_1 = 0, \frac{N}{2*P} - 1$
      /*Read the $i_1$-th pair of rows into $M(0 : 2 * N - 1)$*/
      read($LAF\_A(p), 2 * N, M_2(0), 2 * i_1 * N$)
      /*Perform stride permutation on the two rows*/
      $M(0 : 2 * N - 1) = L_{(N,2)}(M(0 : 2 * N - 1))$
      /*Store the rows into appropriate LAF's*/
      $j_1 = f(2 * i_1 + (\frac{N}{P}) * p);\quad j_2 = f(2 * i_1 + (\frac{N}{P}) * p + 1);$
      $lf_1 = j_1/(\frac{N}{P});\quad lf_2 = j_2/(\frac{N}{P})$
      write($LAF\_B(lf_1), N, M(0), (j_1 - (\frac{N}{P}) * lf_1) * N$);
      write($LAF\_B(lf_2), N, M(N), (j_2 - (\frac{N}{P}) * lf_2) * N$);
   enddo
   Interchange $LAF\_A$ and $LAF\_B$
enddo
```

Figure 4.10: Out-of-core Matrix Transposition node program for node p.

4.16.3 Requirements for HPF-2

Desired extensions

Out-of-core matrix transposition involves working with fixed size data structures with sizes larger than the collective memory of the parallel machine. The provision of the following features would facilitate the expression and compilation of out-of-core algorithms in HPF.

1. Declaration of out-of-core data structures: These declarations would provide information about the creation of file(s) in external memory (see Section 2.3.3).

2. Control over stripe size for distributing files: For the out-of-core matrix transposition algorithm proposed, the basic unit of transfer between main and external memory is a row of the matrix. Storing the elements of a row contiguously would lead to an exploitation of spatial locality (Section 2.3.2).

3. Control over the data distribution of the out-of-core data structures.

Context of extensions

The sequential version of the out-of-core algorithm was presented in [58], in the context of the tensor product project at The Ohio State University.

4.17 FFT: Fast Fourier Transform (VIEWAS)

This section was contributed by C.-H. Huang, P. Sadayappan and S. K. Gupta; all of Ohio State University ({chh,saday,sandeep}@cis.ohio-state.edu).

4.17.1 Application Description

The underlying problem

The FFT is an $O(Nlog(N))$ algorithm to compute the matrix-vector product $y = F\_Nx$, where F_N is the $N \times N$ discrete Fourier transform (DFT) matrix. In the literature, many FFT algorithms have been proposed. These algorithms can be broadly classified into multiplicative FFT algorithms and additive FFT algorithms. The class of additive FFT algorithms contains algorithms which are variations of the Cooley-Tukey FFT algorithm, e.g., the Pease FFT, Korn-Lambiotte FFT, and Stockham FFT [VanL92]. On the other hand, multiplicative FFTs are derivatives of the Good-Thomas FFT. The additive FFTs are more popular than the multiplicative FFT algorithms since they are simpler to implement. We only consider additive FFTs here. Henceforth, an FFT algorithm will refer to an additive FFT algorithm. Furthermore, we only consider 1-D FFT since multidimensional FFTs can be implemented as 1-D FFTs along each of the dimensions of the input array.

Favored parallel algorithm

On a distributed-memory machine, transpose/redistribution based FFTs have been found to perform better than FFT implementations using point-to-point communication. This is because transpose/redistribution based FFT requires a lower communication volume than the FFT with point-to-point communication (for sufficiently large N). With a transpose-based FFT, the input and output array are viewed logically as 2-D arrays. For performing a 1-D FFT of size $N = N_1 \times N_2$, one way to perform a transpose-based FFT is as follows:

1. Reshape the input 1-D array x as an $N_1 \times N_2$ 2-D array. Distribute it according to some $(*, cyclic(b))$ distribution,
2. Transpose array x,
3. Perform FFT along the columns of x^T locally on each processor,
4. Transpose array x,
5. Perform FFT along the columns of x locally on each processor, and
6. Transpose array x to obtain the result.

Note that the above transpose-based FFT requires three transpositions, which amounts to three redistributions. As redistribution involves an all-to-many communication pattern, it is important to minimize the number of redistributions to reduce the communication overhead. As shown in [VanL92], a transposition-based FFT requiring only a single transposition can also be developed.

Redistribution based FFTs have been presented in [GHJS93]. It is shown that using the Stockham FFT algorithm, the 1-D FFT can be implemented using a single redistribution step. This is possible because the bit-reversal permutation is embedded in the computation steps of the Stockham FFT algorithm and therefore does not require the extra bit-permutation step needed in other FFT algorithms. Note that the last transposition in the above FFT implementation is due to bit-reversal. The single redistribution FFT algorithm assumes that $N \geq P^2$. In general, a 1-D FFT can be implemented using a single redistribution whenever N is a multiple of P^2. For simplicity assume that $N = 2^m$ and $P = 2^q$. Then, a 1-D FFT can be performed as follows:

1. Distribute the array x cyclically,

2. Perform the first $m - q$ steps of the computation locally on each processor,

3. Redistribute the array to a *cyclic*(P) distribution,

4. View the distribution of array x as a *cyclic* distribution, and

5. Perform the remaining q steps of the computation locally on each processor.

Step 4 involves changing the view of the distribution from *block* to *cyclic*, without actually moving the data elements, as described in Section 2.4.4. An implicit assumption for the change of the view operation is that the number of elements on each processor are the same under both the old and the new distributions.

The parallelism in Step 2 and Step 5 can be expressed in HPF-1 using the `INDEPENDENT` directive. However, the HPF `INDEPENDENT` directive only asserts that iterations of a loop do not interact or interfere with each other. To generate code for a distributed-memory machine, an HPF compiler also needs to decide on which processor each iteration of the loop should be executed. The mapping of the loop iterations to the processors is important as it has implications on the communication requirements of the resulting code. In general, an `INDEPENDENT` loop may require some initial communication to copy values of data elements used within the iteration and some final communication to copy back the values of non-local data which were modified within the iteration. However, in some cases this communication could be reduced or even eliminated by mapping iterations to the processors in a manner which localizes all or most of the data accessed within each iteration. The presence of complex indexing functions in the loop body, as in the case of FFT, may hinder such optimizations. Hence, there is a need for a mechanism for the programmer to specify the mapping of iterations to processors.

4.17.2 Workarounds for HPF-1

Except for the step involving change of the view of distribution, all other steps in the Stockham FFT algorithm can be directly expressed in HPF-1. The view changing step can be expressed by an array copy. For example, the pseudo-code for the array copy corresponding to 1-D array x with distribution d_1 viewed as having distribution d_2 is as follows:

```
!HPF$ PROCESSOR PROC(P)
!HPF$ DISTRIBUTE (d1) DYNAMIC ONTO PROC :: x
!HPF$ DISTRIBUTE (d1) ONTO PROC :: y
      y(1 : N) = x(1 : N)
!HPF$ REDISTRIBUTE (d2) ONTO PROC :: x
!HPF$ INDEPENDENT NEW i
      DO p = 0, P - 1
         DO i = 0, num_of_local_elements
            x(local_to_global(d2, p, i)) = y(local_to_global(d1, p, i))
         END DO
      END DO
```

where `local_to_global(d1,p,i)` return the global index corresponding to the local index `i` on processor `p`.

4.17.3 Requirements for HPF-2

Desired extensions

VIEWAS. It would be desirable for HPF to provide a construct for changing the view of the distribution of an array. The VIEWAS construct could have the following syntax:

!HPF$ VIEWAS <*target-distribution*> :: <*array-name*>

The semantics of the VIEWAS construct is that the distribution of the array <*array-name*> is henceforth treated to be <*target-distribution*>. No communication is required. On a single processor, the VIEWAS directive has no effect.

Locality Assertion. The information about how the iterations should be mapped to processors can, for example, be provided by using an ON clause along with an INDEPENDENT directive:

!HPF$ INDEPENDENT ON < proc_array_name > (< map_func >) LOCAL,

where < map_func > is a mapping of the associated loop index to the processor index of processor array < proc_array_name >. The above directive specifies that the iterations of the following loop do not interfere with each other and mapping the i-th iteration to processor PROC(k), where k is the value of < map_func > when the value of index variable is i, results in communication-free target code.

Example pseudo-code

Using the VIEWAS construct and locality assertions, the radix-2 Stockham FFT can be expressed as follows:

```
C     Size of FFT = $2^m$, Nos. of proc. = $2^q$, Assumption $m \geq 2q$.
!HPF$ PROCESSOR PROC($2^q$)
!HPF$ DISTRIBUTE (CYCLIC) DYNAMIC ONTO PROC :: A
      COMPLEX * 8  A($0 : 2^m - 1$), B($0 : 2^{m-q} - 1$), omg
      INTEGER i, k, s, m, p, q
      !HPF$ INDEPENDENT NEW (i, k, s, B, omg) ON PROC(p) LOCAL
            DO p = $0, 2^q - 1$
              DO i = $1, m - q$
               DO s = $0, 2^{i-1} - 1$
C     omega(N, k) = $e^{(2\pi\sqrt{-1}k)/N}$
                omg = omega($2^i$, s)
                DO k = $0, 2^{m-q-i} - 1$
                 B($s * 2^{m-i} + k * 2^q + p$) = A($s * 2^{m-i+1} + k * 2^q + p$)
                 B($s * 2^{m-i} + k * 2^q + 2^{m-i+1} + p$) = A($s * 2^{m-i+1} + k * 2^q + 2^{m-i} + p$) * omg
               END DO END DO
               A($p : 2^{m-1} - 2^q + p : 2^q$) = B($p : 2^{m-1} - 2^q + p : 2^q$) + B($p + 2^{m-1} : 2^m - 2^q + p : 2^q$)
               A($p + 2^{m-1} : 2^m - 2^q + p : 2^q$) = B($p : 2^{m-1} - 2^q + p : 2^q$) - B($p + 2^{m-1} : 2^m - 2^q + p : 2^q$)
              END DO
            END DO
!HPF$ REDISTRIBUTE (CYCLIC($2^q$)) :: A
!HPF2$ VIEWAS (CYCLIC) :: A
```

```
!HPF$ INDEPENDENT NEW (i,k,s,B,omg) ON PROC(p) LOCAL
      DO p = 0,2^q - 1
        DO i = m - q + 1,m
          DO s = 0,2^(i-q-1) - 1
            omg = omega(2^i,2^q * s + p)
            DO k = 0,2^(m-i) - 1
              B(s * 2^(m-i) + k * 2^q + p) = A(s * 2^(m-i+1) + k * 2^q + p)
              B(s * 2^(m-i) + k * 2^q + 2^(m-i+1) + p) = A(s * 2^(m-i+1) + k * 2^q + 2^(m-q-i) + p) * omg
          END DO END DO
          A(p : 2^(m-1) - 2^q + p : 2^q) = B(p : 2^(m-1) - 2^q + p : 2^q) + B(p + 2^(m-1) : 2^m - 2^q + p : 2^q)
          A(p + 2^(m-1) : 2^m - 2^q + p : 2^q) = B(p : 2^(m-1) - 2^q + p : 2^q) - B(p + 2^(m-1) : 2^m - 2^q + p : 2^q)
        END DO
      END DO
```

In the above code, communication is required only to perform the redistribution. All the computation before and after the redistribution can be performed locally if iteration p is mapped to processor p.

Context of extensions

A programming environment for automatically synthesizing parallel/vector programs for block recursive algorithms such as FFT and Strassen's matrix multiplication has been developed at the Ohio State University. The system is called EXTENT, *EX*pert system for *TEN*sor product *T*ranslation [DGKL+94]. Programs for redistribution based FFTs can be synthesized automatically by EXTENT.

4.18 SpLU – Sparse LU Factorization

This section was provided by R. Asenjo, M. Ujaldon and E. L. Zapata. Dpt. of Computer Architecture. University of Málaga (SPAIN).

4.18.1 Application Description

The underlying problem

The solution of linear systems $Ax = b$, where matrix A has a sparse sort and huge dimensions, plays a basic role in many fields of science, engineering and economy.

Along this application, we assume a singular sparse matrix A with dimensions $n \times n$ (in the whole matrix there will only be α nonzero elements, such that $\alpha \ll n^2$). By exploiting the sparsity of A, we can reduce significantly the execution time and the memory required to solve the linear system.

There are several methods for solving sparse linear systems [31], [104]. One of them is based on the LU factorization of A [47]. As output of such a factorization, we obtain a couple of matrices, L (lower triangular) and U (upper triangular), with dimensions $n \times n$, and the permutation vectors, π y ρ, with dimension n, such that:

$$A_{\pi_i,\rho_j} = (LU)_{ij} \qquad \forall i,j,\ \ 0 \leq i,j < n. \tag{4.1}$$

The permutation vectors, π y ρ, are needed due to the permutation process that takes place during the factorization in rows and columns of A, with the aim of preserving the sparsity rate and ensuring the numerical stability.

The linear system $Ax = b$ may be solved in five stages:

1. Factorize A giving the L and U matrices and the corresponding permutation vectors, π y ρ.
2. Permute b according to $d_i = b_{\pi_i}$, $0 \leq i < n$, obtaining the d vector.
3. Solve the system $Ly = d$ giving the y vector. This lower triangular system is solved by means of forward-substitution.
4. Solve the system $Uz = y$, in order to get the z vector. This upper triangular system is solved by using back-substitution.
5. Permute z according to $x_{\rho_j} = z_j$, $0 \leq j < n$, obtaining x, the solution vector.

In this application, we will focuse our efforts on the algorithm that computes the first phase (LU factorization) [1]. Sequential algorithms already developed for the sparse LU factorization, like MA28 [30] or Y12M [96], perform several iterations, each of them involving a pivot search in the reduced matrix, followed by a row and column swapping, and an update of range one in the *reduced matrix* (defined as a submatrix containing $(n-k) \times (n-k)$ elements A_{ij}, such that $k \leq i, j < n$, in the $k-th$ iteration). The choice of this pivot must be done in such a way that, on the one hand we preserve the sparsity rate, and, on the other, the numerical stability is guaranteed.

The more widely heuristic strategy used for finding pivots that preserve the sparsity rate is known as Markowitz's strategy [65]. When we choose a pivot, A_{ij}, it may create $M_{ij} = (R_i - 1)(C_j - 1)$ new nonzero elements in the worst case, where R_i (C_j) means the number of nonzero elements in the $i-th$ row $(j-th$ column). The upper bound M_{ij} is known as *Markowitz count* [31, Chap. 7]. The pivot will be chosen such that minimizes the Markowitz count.

Moreover, we must also ensure the numerical stability. In order to accomplish it, we must avoid the selection of pivot elements with low absolute value; that's why we will only accept candidate pivots, A_{ij}, fulfilling:

$$|A_{ij}| \geq u \cdot \max{}_l |A_{lj}|, \tag{4.2}$$

where u, $0 \leq u \leq 1$ is a threshold parameter [31, Chap. 7].

In this work, we show a parallel algorithm for the LU factorization of generic sparse matrices on a distributed memory multiprocessor with mesh topology ($P \times Q$ processors). This factorization was broach by Stappen et al. on a network of transputers [94]. The parallel code is SPMD (Simple Program Multiple Data), and PVM [64] is used as a message passing interface. The data distribution follows the philosophy of the scatter methods. Each processor stores the nonzero elements belonging to its local matrix like a semi-ordered, two-dimensional linked list [94]. With all these features, our code was implemented on the CRAY T3D machine [68].

Favored parallel algorithms

An algorithm for distributed memory systems is the Stappen et al. [94]. Such an algorithm distinguishes different phases: Search for pivots, rows and columns permutations, and updating of the reduced submatrix. They also perform a detailed study of the algorithm complexity associated to

[1] Nevertheless, we hope to achieve results for the whole algorithm for the final version of the application.

the local storage structure of the data. The conclude claiming that the unordered two-dimensional doubly linked-list yields minimum execution times, according to the strategy of permuting rows and columns in the submatrix. Experimental results over a mesh up to 400 transputers, show speedups up to 107.

We present here an algorithm for the sparse LU decomposition on distributed memory multicomputers, as Stappen et al. Our local storage structure is also a doubly linked list, but sorted by rows and disordered by columns, what achieves less complexity and updates the matrix directly over the data structure. Likewise, a new data distribution as the BRS [93] will let us to focus the problem from a point of view closer to the data-parallel programming. At the end of this work, experimental results are showed on the CRAY T3D machine [6].

Detailed algorithm

The parallel algorithm executes a number of iterations, each containing three different phases: Pivots search, rows and columns permutation, and reduced matrix and R and C vectors update.

The parallelism inherent to the sparse LU algorithm involves two issues. First of all, in the dense case, we can parallelize the loops traversing all the updating of range one in the reduced matrix. This parallelism is inherited by the sparse algorithm. Secondly, the sparse algorithm allows us to perform parallel computations that must be sequentialized in the dense case. Thus, in the sparse algorithm, it is possible to merge several updates of range one in a single update process of multiple range (m) by modifying the Markowitz strategy in such a way that we search for a pivot set containing m compatible pivots (referred to as *PivotSet*). Two pivots, A_{ij} and A_{kl} are compatible and independent if

$$A_{il} = A_{kj} = 0. \tag{4.3}$$

A relaxation in the compatibility criteria can be considered if we only force $A_{kj} = 0$. In this case, both pivots are called *partially compatible*, what increases the size of *PivotSet* sets, whereas the update process is more difficult to carry out. In Figure 4.11, it is shown the different types of compatible pivots.

A description of the complete algorithm follows:

Parallel Sparse LU Algorithm

```
$\pi$ = id;
$\rho$ = id;
initialize $R$ and $C$;
$k = 0$;
while ( $k < n$ )
    {
      find pivot set $PivotSet = (i_r, j_r) \; : \; 0 \leq r < m$;
      permute rows, $\pi_i$ and $R_i$;
      permute columns, $\rho_j$ and $C_j$;
      update matrix elements, $A$;
      update $R$ and $C$ nonzero counts;
      $k = k + m$;
    }
```

In the description of the algorithm showed above, we implicitly assume that each processor (s, t) performs computations over its local data. At the end of the algorithm, matrix A will have

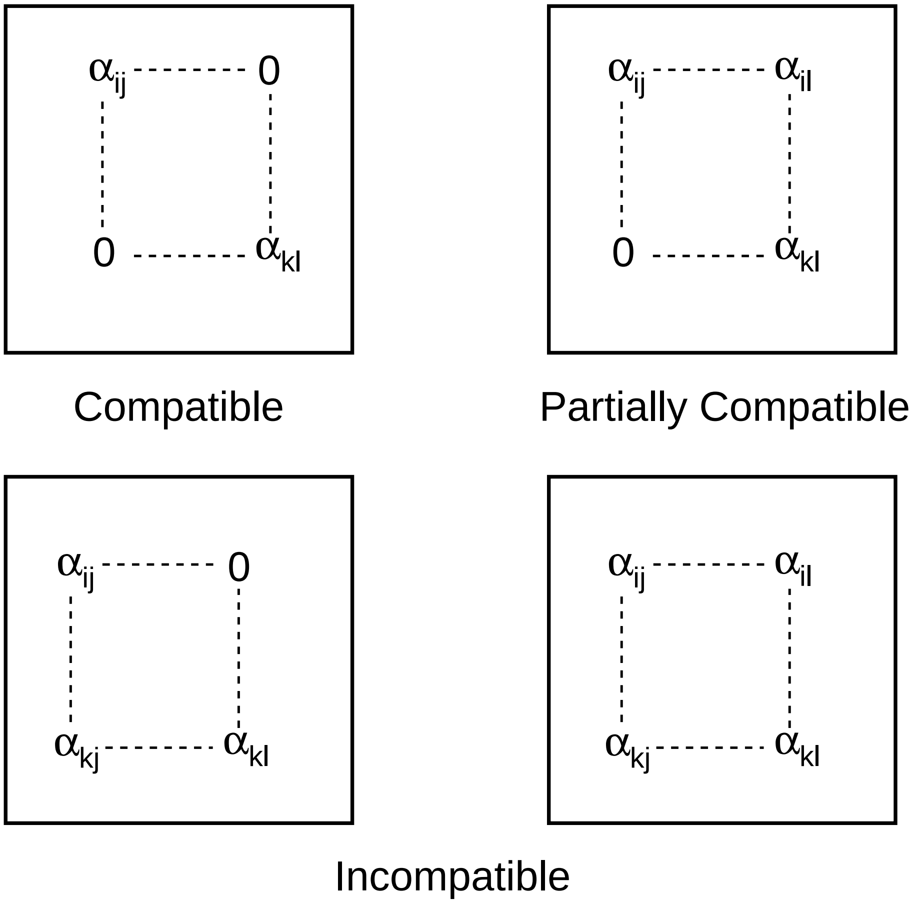

Figure 4.11: Types of compatibility.

the L factor in its strictly lower triangular matrix, and the U factor in its upper triangular matrix. Vectors π and ρ will store the final values. This parallel algorithm is based on a sequential one that exhibits the same structure already described.

In Figure 4.12, we show an example of a sparse matrix with $n = 8$ and $\alpha = 13$.

Data structures and layout

To minimize the number of communications, we have tried to pack all the messages as many as possible, thus reducing the communication latency time. As well, to decrease the number of messages and duplicate the speed in the broadcast and reduce operations, messages are broadcasted (reduced) from (to) the center of the rows, columns or mesh.

Following this approach, as an example, we accumulate all the incompatible pivots in the processor $(P/2, Q/2)$, where we create *PivotSet* and, subsequently, we broadcast it with a complexity of $\mathcal{O}((P+Q)/2)$. Note that Stappen et al. create the PivotSet by considering a pipeline in the first row of the mesh, in which all the pivots are passing one by one through the mesh, from processor $(0, t)$; $0 \leq t < Q$ to processor $(0, Q-1)$, creating in this last processor the *PivotSet*. This alternative increase the total number of messages and the overhead associated to the latencies.

In relationship with the local data storage, we can also choose another structure that reduces

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 0 & 2 & 10 \\ 0 & 11 & 3 & 0 & 0 & 12 & 13 & 0 \end{pmatrix}$$

Figure 4.12: Sparse matrix with $n = 8$ and $a = 13$

the wastage of memory with respect to the linked-list one. We will name it Block Row Scatter (BRS) [93]. Lets assume the matrix A partitioned into a set of submatrices $B(l,k)$ with dimensions $P \times Q$, such that $A_{ij} =^{kl} B_{st}$ where $i = k \cdot P + s,\ j = l \cdot Q + t\ \ (0 \leq i,j < n)$. Pairs (i,j), (s,t) and (k,l) represent global, local and block indices, respectively. In order to perform the partition of matrix A giving submatrices $B(k,l)$, it may be necessary to add new rows or columns containing null elements to it. The distribution of the elements of matrix A is performed by mapping each one of the blocks of size $P \times Q$ onto the mesh of processors. The data storage format consists of three vectors, D, C y R. The D vector stores the nonzero values of the matrix, as they are traversed in a row-wise fashion. The C vector contains the block column indices of the elements in the D vector. Finally, the R vector stores the indices in the D vector that correspond to the first non-zero element of each row. By convention, we define one additional element in R vector with the value of the number of elements in D plus one. The memory spent is drastically reduced, although the access to the data by columns may be slowed down. Figure 4.13 show the BRS local storage for the matrix of the figure 4.12 onto a 2×2 mesh.

$$\#0 \quad \overset{D}{\begin{pmatrix} 1 \\ 2 \\ 9 \\ - \end{pmatrix}} \overset{C}{\begin{pmatrix} 1 \\ 4 \\ 3 \\ - \end{pmatrix}} \overset{R}{\begin{pmatrix} 1 \\ 3 \\ 3 \\ 3 \\ 4 \end{pmatrix}} \qquad \#1 \quad \overset{D}{\begin{pmatrix} 5 \\ 7 \\ 10 \\ - \end{pmatrix}} \overset{C}{\begin{pmatrix} 4 \\ 2 \\ 4 \\ - \end{pmatrix}} \overset{R}{\begin{pmatrix} 1 \\ 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}}$$

$$\#2 \quad \overset{D}{\begin{pmatrix} 3 \\ 4 \\ 6 \\ 13 \end{pmatrix}} \overset{C}{\begin{pmatrix} 2 \\ 4 \\ 3 \\ 4 \end{pmatrix}} \overset{R}{\begin{pmatrix} 1 \\ 3 \\ 4 \\ 4 \\ 5 \end{pmatrix}} \qquad \#3 \quad \overset{D}{\begin{pmatrix} 8 \\ 11 \\ 12 \\ - \end{pmatrix}} \overset{C}{\begin{pmatrix} 1 \\ 1 \\ 3 \\ - \end{pmatrix}} \overset{R}{\begin{pmatrix} 1 \\ 1 \\ 1 \\ 2 \\ 4 \end{pmatrix}}$$

Figure 4.13: BRS storage.

4.18.2 Workarounds for HPF-1

The sparse LU algorithm presented above is a parallelization by hand that increases the inherent parallelism of the algorithm by grouping several individual steps in a ”diagonal block” step capable of parallelization. Such an strategy requires to permute rows and columns in the sparse matrix before entering the update loop, what demands a dynamic data type as double linked lists to embed or remove new elements.

This kind of algorithm cannot be implemented in HPF because of the handling, partitioning and distribution of dynamic data structures. However, besides this algorithm, there is another alternative that, although neither can be implemented in HPF, it is closer to what a data-parallel language expects as input.

This alternative is the one we show below. It computes N steps in the main loop, instead of less steps involving diagonal blocks. In this way, we are losing parallel performance, but the code is straightforward. We have used new data-parallel extensions we have designed for sparse and irregular computation, including specific representations (as the CCS - Compressed Column Storage format used below) and distributions (as the BRS - Block Row Storage used below too) for sparse problems. These elements have been included into the Vienna-Fortran language and the compiler that accepts them is currently under development [92].

```
C HPF algorithm with new Vienna-Fortran sparse directives (highlighted
C with *VF*) for computing the LU decomposition in an NxN sparse matrix
C with "ALPHA" nonzero elements.

      PARAMETER(X=NUMBER_OF_PROCESSORS(DIM=1))
      PARAMETER(Y=NUMBER_OF_PROCESSORS(DIM=2))
!HPF$ PROCESSORS MESH(X,Y)

      PARAMETER (ALPHA=20, N = 10)
      INTEGER I, J, K, L, M1, M2
*VF*  REAL A(N,N), SPARSE(CCS(DA, CA, RA)), DYNAMIC

*VF*  DISTRIBUTE A :: (CYCLIC, CYCLIC)

      DO I = 1, N-1
        J = 0
        DO WHILE ((RA(CA(I)+J) .LT. I) .AND. (J .LE. (CA(I+1)-CA(I))))
          J = J + 1
        ENDDO
        IF (J .GT. (CA(I+1)-CA(I)))
          WRITE(*,*) "ERROR: ZERO IN THE DIAGONAL ELEMENT"
          STOP
        ENDIF
        Diag = DA(CA(I)+J)  ! Always must be an element in the diagonal
!HPF$   INDEPENDENT
        DO K = J+1, CA(I+1)-CA(I)  ! Update all the elements below diagonal
          DA(CA(I)+K) = DA(CA(I)+K) / Diag
        ENDDO
!HPF$   INDEPENDENT
```

```
        DO K = J+1, CA(I+1)-CA(I) ! Traverse the rows with items in the i-th col
!HPF$     INDEPENDENT
          DO L = I+1, N  ! Traverse all the colums to find items in the i-th row
            M1 = 0
            DO WHILE ((M1 .LE. (CA(L+1)-CA(L))) .AND. (RA(CA(L)+M1) .LT. I))
              M1 = M1 + 1
            ENDDO
            IF (M1 .LE. (CA(L+1)-CA(L))) THEN  ! No items in this subcolumn
              IF ((RA(CA(L))+M1+1) .EQ. I) THEN  ! Found a item to make a pair
                M2 = M1+2
                DO WHILE ((M2 .LE. (CA(L+1)-CA(L))) .AND.
                          (RA(CA(L)+M2) .LT. RA(CA(I)+K)))
                  M2 = M2 + 1
                ENDDO
                IF (M2 .LE. (CA(L+1)-CA(L))) THEN
                  IF (RA(CA(L)+M2) .EQ. RA(CA(I)+K)) THEN ! Update existing item
                    DA(CA(L)+M2) = DA(CA(L)+M2) - DA(CA(I)+K) * DA(CA(L)+M1+1)
                  ELSE
                    The element isn't in the matrix => Fill-in between two
                                                       elements in this row.
                    Fill-in in dense coordinates A(RA(CA(I)+K,L)
                  ENDIF
                ELSE
                  The element isn't in the matrix => Fill-in after last
                                                     item of this row.
                  Fill-in in dense coordinates A(RA(CA(I)+K,L)
                ENDIF
              ENDIF
            ENDIF
          ENDDO
        ENDDO
      ENDDO
END
```

4.18.3 Requirements for HPF-2

Since the code showed is going to be accepted as input to our sparse compiler implemented within the Vienna Fortran Compilation System, we proof that the sparse extensions already mentioned suffice to express this algorithm by using a data-parallel language, either Vienna-Fortran or HPF. Also, another examples of sparse codes has been successfully expressed in Vienna-Fortran by using our new language elements:

- Sparse Matrix Vector Product,
- Sparse Matrices Multiplication.
- The Lanczos Algorithm to tridiagonalize a sparse matrix or compute its eigenvalues.
- The Conjugate Gradient method, a well-known iterative algorithm for solving linear systems.

Bibliography

[1] G. Agrawal and J. Saltz. Interprocedural communication optimizations for distributed memory compilation. In *Proc. Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[2] Gagan Agrawal, Alan Sussman, and Joel Saltz. Compiler and runtime support for structured and block structured applications. In *Supercomputing 93*, pages 578–587, November 1993. An extended version available as University of Maryland Technical Report CS-T R-3052 and UMIACS-TR-93-29.

[3] Gagan Agrawal, Alan Sussman, and Joel Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Trans. on Parallel and Distributed Systems*, 1994. To appear. Also available as University of Maryland Technical Report CS-TR-3143 and UMIACS-TR-93-94.

[4] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. SIGPLAN '93 Conf. on Program Language Design and Implementation*, Albuquerque, NM, June 1993.

[5] Applications Working Group Of The Scalable I/O Initiative. Preliminary survey of I/O intensive applications. Technical Report CCSF-38, Concurrent Supercomputing Consortium, Caltech, Pasadena, CA 91125, January 1994. Scalable I/O Initiative Working Paper No. 1.

[6] R. Asenjo and E. L. Zapata. Sparse LU factorization on the CRAY T3D. Submitted to: High-Performace Computing and Networking Europe'95, May 1995.

[7] C. Ashcraft, S. Eisenstat, and J. W.-H. Liu. A fan-in algorithm for distributed sparse numerical factorization. SIAM J. Sci. Statist. Comput., 11:593–599, 1990.

[8] C. C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon. Progress in sparse matrix methods for large linear systems on vector supercomputers. *Internat. J. Supercomputer Appl.*, 1:10–30, 1987.

[9] D. H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4:23–35, 1990.

[10] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[11] S. Benkner. *Vienna Fortran 90 and its Compilation.* PhD thesis, technical University of Vienna, Vienna, Austria, September 1994.

[12] R. Bordawekar, A. Choudhary, K. Kennedy, and C. Koelbel. Models and compilation strategies for out-of-core data parallel prog rams. Technical Report Under Preperation, Northeast Parallel Architectures Center, Syracuse University, and CRPC, 1994.

[13] R. Bordawekar, A. Choudhary, and R. Thakur. Data Access Reorganizations in Compiling Out-of-core Data Parallel Programs on Distributed Memory Machines. Technical Report SCCS–622, NPAC, Syracuse University, April 1994.

[14] R. Bordawekar, J. del Rosario, and A. Choudhary. Design and Evaluation of Primitives for Parallel I/O. In *Proc. Supercomputing'93*, pages 452–461, November 1993.

[15] P. Brezany, M. Gerndt, P. Mehrotra, and H. Zima. Concurrent File Operations in a High Performance Fortran. In *Proc. Supercomputing '92*, pages 230–238, November 1992.

[16] P. Brezany, M. Gerndt, V. Sipkova, and H. Zima. SUPERB support for irregular scientific computations. In *Proc. 1992 Scalable High Performance Computing Conf.*, Williamsburg, VA, April 1992.

[17] S. Chakrabarti and K. Yelick. Implementing an irregular application on a distributed memory multiprocessor. In *Proc. Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[18] K. M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng. Integrated support for task and data parallelism. *Intl. J. Supercomputer Applications*, 2(8):80–98, 1994.

[19] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran - a Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[20] Barbara M. Chapman, Piyush Mehrotra, John Van Rosendale, and Hans P. Zima. A software architecture of multidisciplinary applications: Integrating task and data parallelism. In *Proceedings of CONPAR94-VAPP VI Third International Conference on Vector and Parallel Processing, LNCS 854*, pages 664–676. Springer Verlag, September 1994.

[21] A. Choudhary, R. Bordawekar, M. Harry, R. Krishnaiyer, R. Ponnusamy, T. Singh, and R. Thakur. PASSION: Parallel and Scalable Software for Input-Output. Technical Report SCCS–636, NPAC, Syracuse University, September 1994.

[22] P. Corbett, D. Feitelson, J. Prost, and S. Baylor. Parallel Access to Files in the Vesta File System. In *Proc. Supercomputing '93*, pages 472–481, November 1993.

[23] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural requirements of parallel scientific applications with explicit communication. In *Proc. of Intl. Symposium on Computer Architecture*, pages 2–13, 1993.

[24] E. D. Dahl. Mapping and compiled communication on the connection machine system. In *Proc. 5th Distributed Memory Computing Conf.*, Charleston, SC, April 1990.

[25] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. In *Proc. 6th Workshop on Languages and Compilers for Parallel Computing*, pages 152–168. Springer-Verlag, August 1993. Also available as University of Maryland Technical Report CS-TR-3076 and UMIACS-TR-93-42.

[26] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994. Also available as University of Maryland Technical Report CS-TR-3163 and UMIACS-TR-93-109.

[27] J. del Rosario and A. Choudhary. High performance i/o for parallel computers: Problems and prospects. *IEEE Computer*, March 1994.

[28] J. del Rosario, M. Harry, and A. Choudhary. The Design of VIP-FS: A Virtual Parallel File System for High Performance Parallel and Distributed Computing. Technical Report SCCS-628, NPAC, Syracuse University, May 1994.

[29] J. M. del Rosario and A. Choudhary. High-performance I/O for massively parallel computers — problems and prospects. *IEEE Computer*, pages 59–68, 1994.

[30] I. S. Duff and J. K. Reid. Some design features of a sparse matrix code. *ACM Trans. Math. Software*, 5:18–35, 1979.

[31] I. S. Duff A. M. Erisman and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, Oxford, U.K., 1986.

[32] K. Eswar, C.-H. Huang, and P. Sadayappan. Memory-adaptive parallel sparse cholesky factorization. In *Proceedings of the 1994 Scalable High-Performance Computing Conference '94*, pages 317–323, Knoxville, TN, 1994.

[33] K. Eswar, C.-H. Huang, and P. Sadayappan. On mapping data and computation for parallel sparse cholesky factorization. Technical Report OSU-CISRC-7/94-TR40, Department of Computer and Information Science, The Ohio State University, 1994.

[34] K. Eswar, P. Sadayappan, C.-H. Huang, and V. Visvanathan. Supernodal sparse cholesky factorization on distributed-memory multiprocessors. In *Proceedings of the Twenty-second International Conference on Parallel Processing*, volume III, pages 18–22, St. Charles, IL, 1993.

[35] K. Eswar, P. Sadayappan, and V. Visvanathan. Multifrontal factorization of sparse matrices on shared-memory multiprocessors. In *Proceedings of the Twentieth International Conference on Parallel Processing*, volume III, pages 159–166, St. Charles, IL, 1991.

[36] K. Eswar, P. Sadayappan, and V. Visvanathan. Parallel direct solution of sparse linear systems. In F. Özgüner and F. Erçal, editors, *Parallel Computing on Distributed Memory Multiprocessors*, Berlin and Heidelberg and New York, 1993. Springer Verlag.

[37] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report Computer Science Dept. Technical Report CS-94-230, University of Tennessee, April 1994. To appear in the International Journal of Supercomputer Applications, Volume 8, Number 3/4, 1994.

[38] I. Foster, B. Avalani, A. Choudhary, and M. Xu. A compilation system that integrates High Performance Fortran and Fortran M. In *Proc. 1994 Scalable High Performance Computing Conf.*, pages 293–300, 1994.

[39] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 25(1), 1995.

[40] I. Foster, J. Tilson, A. Wagner, R. Shepard, R. Harrison, R. Kendall, and R. Littlefield. High performance computational chemistry: (I) Scalable Fock matrix construction algorithms. Preprint, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1994.

[41] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[42] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[43] W. M. Gentleman and G. Sande. Fast Fourier transforms for fun and profit. In *Proc. AFIPS*, volume 29, pages 563–578, 1966.

[44] A. George, M. Heath, J.W.-H Liu, and E.G.-Y. Ng. Sparce Cholesky factorization on an local-memory multiprocessor. *SIAM J. Sci. Statist. Comput.*, 9:327–340, 1988.

[45] A. George and J.W.-H. Liu. *Computer Solution of Large Sparce Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[46] H. M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Bonn, December 1989.

[47] G. H. Golub and C. F. Van Loan. *Matrix Computation.* The Johns Hopkins University Press, Baltimore, MD, 2 edition, 1989.

[48] A. Gupta and V. Kumar. A scalable parallel algorithm for sparse matrix factorization. Technical Report TR 94-19, Department of Computer Science, University of Minnesota, 1994.

[49] Manish Gupta, Edith Schonberg, and Harini Srinivasan. A unified data-flow framework for optimizing communication. In *Proc. Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, New York, August 1994.

[50] R. v. Hanxleden, K. Kennedy, and J. Saltz. Value-based distributions in fortran d — a preliminary report. Technical Report CRPC-TR93365-S, Center for Research on Parallel Computation, Rice University, December 1993. submitted to Journal of Programming Languages - Special Issue on Compiling and Run-Time Issues for Distributed Address Space Machines.

[51] R. Harrison et al. High performance computational chemistry: (II) A scalable SCF code. Preprint, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1994.

[52] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

[53] W. Hehre, L. Radom, P. Schleyer, and J. Pople. *Ab Initio Molecular Orbital Theory.* John Wiley and Sons, 1986.

[54] High Performance Fortran Forum. High Performance Fortran journal of development. Technical Report CRPC-TR93300, Center for Research on Parallel Computation, Rice University, Houston, TX, May 1993.

[55] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, May 1993.

[56] Holian, Mandell, Adams, Addessio, Baumgardner, and Mosso. Mesa: A 3-d computer code for armor/anti-armor applications. In *Proceedings of the Supercomputing World Conference*, 1990.

[57] Intel. Paragon XP/S System Description. Technical Report Intel Advanced Information, Intel Corporation, 1992.

[58] S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. Efficient transposition algorithms for large matrices. In *Proc. of Supercomputing '93*, pages 656–666, 1993.

[59] C. Koelbel and P. Mehrotra. Programming data parallel algorithms on distributed memory machines using Kali. In *Proc. 1991 ACM International Conf. on Supercomputing*, Cologne, Germany, June 1991.

[60] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machines. In *Proc. Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

[61] Kothe, Baumgardner, Cerutti, Daly, Holian, Kober, Mosso, Painter, Smith, and Torrey. Pagosa: A massively parallel multi-material hydrodynamics model for three-dimensional high-speed flow and high-rate material deformation. In Adrian Tentner, editor, *High Performance Computing 1993: Grand Challenges in Computer Simulation*, pages 9–14. The Society for Computer Simulation, 1993.

[62] D. J. Kuck and A. H. Sameh. A supercomputing performance evaluation plan. In *Proceedings of the International Conference on Supercomputing*, Athens, Greece, June 1987. Springer Verlag. Lecture Notes in Computer Science, #297, pp. 1–17.

[63] B. Lee and F. M. Richards. The interpretation of protein structures: Estimation of static accessibility. *Journal of Molecular Biology*, 55:379–400, 1971.

[64] A. Geist A. Beguelin J. Dongarra W. Jiang R. Manchek and V. Sunderam. *PVM 3 User's Guide and Refernce Manual.* Engineering Physic and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1993.

[65] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Sci.*, 3:255–269, 1957.

[66] G. McRae, W. Goodin, and J. Seinfeld. Development of a second-generation mathematical model for urban air pollution - 1. Model formulation. *Atmospheric Environment*, 16(4):679–696, 1982.

[67] G. McRae, A. Russell, and R. Harley. *CIT Photochemical Airshed Model - Systems Manual.* Carnegie Mellon University, Pittsburgh, PA, and California Institute of Technology, Pasadena, CA, February 1992.

[68] S. Booth J. Fisher N. MacDonald P. Maccallum E. Minty and A. Simpson. *Parallel Programming on the Cray T3D.* Edinburgh Parallel Computing Centre, University of Edinburgh, U.K., September 1994.

[69] Bongki Moon, Gopal Patnaik, Robert Bennett, David Fyfe, Alan Sussman, Craig Douglas, Joel Saltz, and K. Kailasanath. Runtime support and dynamic load balancing strategies for structured adaptive applications. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing.* SIAM, February 1995. To appear.

[70] D. Noll, J. Pauly, C. Meyer, D. Nishimura, and A. Macovski. Deblurring for non 2d-fourier transform magnetic resonance imaging. *Magnetic Resonance in Medicine*, 25:319–333, 1992.

[71] M. Okutomi and T. Kanade. A multiple-baseline stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(4):353–363, 1993.

[72] M. A. Olson and K. D. Kimsey. Calculation of elastic-plastic wave propagation on the connection machine. Technical Report BRL-TR-3360, Balistic Research Laboratory, June 1992.

[73] S. Plimpton, Gary Mastin, and Dennis Ghiglia. Synthetic aperture radar image processing on parallel supercomputers. In *Proc. of Supercomputing '91*, pages 446–452, Albuquerque, NM, November 1991.

[74] Ravi Ponnusamy, Yuan-Shin Hwang, Joel Saltz, Alok Choudhary, and Geoffrey Fox. Supporting irregular distributions in FORTRAN 90D/HPF compilers. Technical Report CS-TR-3268 and UMIACS-TR-94-57, Computer Science Department, University of Maryland, May 1994. To appear in IEEE Parallel and Distributed Technology.

[75] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. Runtime support and compilation methods for user-specified data distributions. Technical Report CS-TR-3194 and UMIACS-TR-93-135, University of Maryland, November 1993. To appear in IEEE Transactions on Parallel and Distributed Systems.

[76] J. Ramanujam and P. Sadayappan. Iteration space tiling for distributed memory machines. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[77] D. F. G. Rault and M. S. Woronowicz. Spacecraft contamination investigation by direct simulation Monte Carlo - contamination on UARS/HALOE. In *Proceedings AIAA 31th Aerospace Sciences Meeting and Exhibit, Reno, Nevada*, January 1993.

[78] A. Reeves and C. Chase. The Paragon programming paradigm and distributed-memory multicomputers. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[79] Timothy J. Richmond. Solvent accessible surface area and excluded volue in proteins. *Journal of Molecular Biology*, 178:63–89, 1984.

[80] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proc. SIGPLAN '89 Conf. on Program Language Design and Implementation*, Portland, OR, June 1989.

[81] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse cholesky factorization. In *Proc. of Supercomputing '92*, Minneapolis, November 1992.

[82] P. Sadayappan and V. Visvanathan. Circuit simulation on shared-memory multiprocessors. *IEEE Transactions on Computers*, C-37(12):1634–1642, December 1988.

[83] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[84] R. Schreiber. A new implementation of sparse gaussian elimination. *ACM Trans. Math. Software*, 8:256–276, 1982.

[85] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. Run-time and compile-time support for adaptive irregular problems. In *Proc. of Supercomputing '94*, November 1994. To appear.

[86] G. Shaw, R. Gabel, D. Martinez, A. Rocco, S. Pohlig, A. Gerber, J. Noonan, and K. Teitelbaum. Multiprocessors for radar signal processing. Technical Report 961, MIT Lincoln Laboratory, November 1992.

[87] A. Shrake and J. A. Rupley. Environment and exposure to solvent of protein atoms. *Journal of Molecular Biology*, 79:351–371, 1973.

[88] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross. Exploiting task and data parallelism on a multicomputer. In *Proc. SIGPLAN '93 Conf. on Program Language Design and Implementation*, Albuquerque, NM, June 1993.

[89] R. Thakur, R. Bordawekar, and A. Choudhary. Compiler and Runtime Support for Out-of-Core HPF Programs. In *Proc. 8th ACM International Conf. on Supercomputing*, pages 382–391, July 1994.

[90] R. Thakur, R. Bordawekar, and A. Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. *Proc. Workshop on I/O in Parallel Computer Systems at IPPS '94*, April 1994.

[91] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnuswamy, and T. Singh. PASSION runtime library for parallel I/O. In *Proc. of Scalable Parallel Libraries Conference*, 1994. to appear.

[92] M. Ujaldon, E.L. Zapata, B. Chapman, and H. Zima. New data-parallel language features for sparse matrix computations. Submitted to 9th International Parallel Processing Symposium. April 1995. Santa Barbara. California.

[93] R. Asenjo L. F. Romero M. Ujaldón and E. L. Zapata. Sparse block and cyclic data distributions for matrix computations. In *High Performance Computing: Technology and Application.* Grandinetti et al. (Eds.) Elsevier Science, (to appear).

[94] A. F. van der Stappen R. H. Bisseling and J. G. G. van de Vorst. Parallel sparse LU decomposition on a mesh network of transputers. *SIAM J. Matrix Anal. Appl.*, 14(3):853–879, July 1993.

[95] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform.* SIAM, Philadelphia, PA, 1992.

[96] Z. Zlatev J. Wasniewski and K. Schaumburg. *Y12M—Solution of Large and Sparse Systems of Linear Algebraic Equations*. Number 121 in Lecture Notes in Computer Science. Springer-Verlang, Berlin, 1981.

[97] R. Weaver and R. Schnabel. Automatic mapping and load balancing of pointer-based dynamic data structures on distributed memory machines. In *Proc. 1992 Scalable High Performance Computing Conf.*, Williamsburg, VA, April 1992.

[98] J. Webb. Implementation and performance of fast parallel multi-baseline stereo vision. In *Computer Architectures for Machine Perception*, pages 232–240, December 1993.

[99] J. Webb. Latency and bandwidth consideration in parallel robotics image processing. In *Proc. of Supercomputing '93*, pages 230–239, November 1993.

[100] M. S. Woronowicz and D. F. G. Rault. On predicting contamination levels of HALOE optics aboard UARS using direct simulation Monte Carlo. In *Proceedings AIAA 28th Thermophysics Conference, Orlando, Florida*, June 1993.

[101] ANSI Technical Committee X3H5. Parallel processing model for high level programming models. Technical report, American National Standards Institute (ANSI), 1992.

[102] B. Yang, J. Webb, J. Stichnoth, D. O'Hallaron, and T. Gross. Do&Merge: Integrating parallel loops and reductions. In *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 169–183, Portland, OR, August 1993. Springer Verlag.

[103] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.

[104] Z. Zlatev. *Computational Methods for General Sparse Matrices, Mathematics and Its Applications*. 65. Kluwer Academin Publisher, Dordrecht, the Netherlands, 1991.