# Supporting Irregular Distributions in FORTRAN 90D/HPF Compilers *

**Ravi Ponnusamy**[†‡] **Yuan-Shin Hwang**[†] **Raja Das**[†]
**Joel Saltz**[†] **Alok Choudhary**[‡] **Geoffrey Fox**[‡]

[†] *UMIACS and Computer Science Department*
*University of Maryland*
*College Park, MD 20742*

[‡] *Northeast Parallel Architectures Center*
*Syracuse University*
*Syracuse, NY 13244*

## Abstract

*This paper presents methods that make it possible to efficiently support irregular problems using data parallel languages. The approach involves the use of a portable, compiler-independent, runtime support library called CHAOS. The CHAOS runtime support library contains procedures that*

- *support static and dynamic distributed array partitioning,*
- *partition loop iterations and indirection arrays,*
- *remap arrays from one distribution to another, and*
- *carry out index translation, buffer allocation and communication schedule generation.*

*The CHAOS runtime procedures are used by a prototype Fortran 90D compiler as runtime support for irregular problems. This paper also presents performance results of compiler-generated and hand-parallelized versions of two stripped down applications codes. The first code is derived from an unstructured mesh computational fluid dynamics flow solver and the second is derived from the molecular dynamics code CHARMM.*

*A method is described that makes it possible to emulate irregular distributions in HPF by reordering elements of data arrays and renumbering indirection arrays. The results suggest that an HPF compiler could use reordering and renumbering extrinsic functions to obtain performance comparable to that achieved by a compiler for a language (such as Fortran 90D) that directly supports irregular distributions.*

*This work was sponsored in part by ARPA (NAG-1-1485), NSF (ASC 9213821), and ONR (SC292-1-22913).

# 1 Introduction

On distributed memory machines, large data arrays need to be partitioned between local processor memories. These partitioned data arrays are called *distributed arrays*. Many applications can be efficiently implemented by using simple schemes for mapping distributed arrays. One example of such a scheme is to divide an array into contiguous, equal sized subarrays and to assign each subarray to a different processor. Another example is to assign consecutively indexed array elements to processors in a round-robin fashion. These two standard data distribution schemes are often called BLOCK and CYCLIC data distributions [10], respectively. Languages such as High Performance Fortran (HPF) [10], Fortran D [6] and Vienna Fortran [4] allow users to control how array elements are assigned to processor memories.

Many scientific applications make extensive use of indirectly accessed arrays. Examples of such problems include computational fluid dynamics codes [11], molecular dynamics codes (CHARMM, AMBER, GROMOS, etc.) [3], diagonal or polynomial preconditioned iterative linear solvers, and time dependent flame modeling codes. These problems are called *irregular problems*. Figure 1 illustrates code with an irregular loop. This example shows the code that sweeps over *nedge* mesh edges. Arrays `x` and `y` are *data arrays*. Loop iteration $i$ carries out a computation involving the edge that connects vertices $edge1(i)$ and $edge2(i)$. Arrays such as `edge1` and `edge2` which are used to index data arrays are called *indirection arrays*.

```
C   Outer Loop L1
    do n = 1, n_step
    ...
C   Inner Loop L2
      do i = 1, nedge
        y(edge1(i)) = y(edge1(i)) + f(x(edge1(i)), x(edge2(i)))
        y(edge2(i)) = y(edge2(i)) + g(x(edge1(i)), x(edge2(i)))
    end do
    ...
    end do
```

Figure 1: An Example Code with an Irregular Loop

It has been widely observed (e.g. [5], [13]) that performance on distributed memory systems can be enhanced by distributing data using a non-standard format. Researchers have developed a variety of methods to obtain data mappings that are designed to optimize irregular problem communication requirements [1, 16, 18]. The distribution produced by these methods needs to be represented by a table that associates a processor assignment with each array element. This kind of distribution is often called an *irregular distribution*.

Figure 2 depicts three different distributions of data arrays over two processors. Figure 2(a) shows the graph of 6 nodes and 7 edges. Arrays `x` and `y` are data arrays. The edges are represented by two indirection arrays `edge1` and `edge2`, which will be partitioned in blocks. The code listed in Figure 1 can be used to sweep this graph. Figure 2(b) presents the result of BLOCK distribution; nodes 1, 2, and 3 are assigned to processor $P0$, and nodes 4, 5, and 6 to processor $P1$. The dashed circles in indirection arrays `edge1` and `edge2` indicate that the indexed elements are not local. The CYCLIC distribution of the graph is displayed in Figure 2(c). Nodes are assigned to processors in round robin fashion. In this distribution, there are 5 non-local data elements. The irregular distribution shown Figure 2(d) represents the best mapping of the graph since only one remote reference is required.

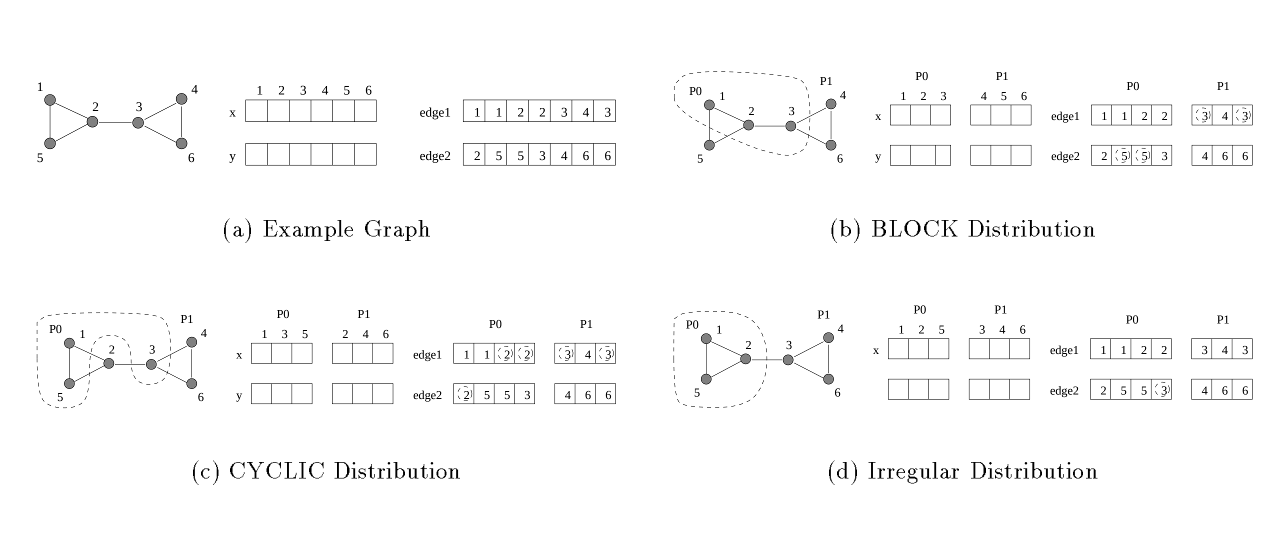


Figure 2: Data Distributions

In addition to standard distributions such as BLOCK and CYCLIC, Fortran D and Vienna Fortran also support irregular data distributions. Fortran D allows a user to explicitly specify an irregular distribution using an array, to specify a mapping of array elements to processors. Vienna Fortran allows user-defined functions to describe irregular distributions. The current version of HPF does not directly support irregular distributions. Language extensions have been proposed by Hanxeleden et al [8] and Ponnusamy et al [12] to support irregular distributions in languages like Fortran D. A method is described in this paper that makes it possible to emulate irregular distributions in HPF by reordering elements of data arrays and renumbering indirection arrays. This paper presents results that suggest that an HPF compiler could use reordering and renumbering extrinsic functions to obtain performance comparable to that achieved by a compiler for a language (such as Fortran 90D) that directly supports irregular distributions. Researchers have proposed compile-time techniques to partition data automatically by compilers. But their approaches are only applied to regular programs [14].

This paper considers two additional language features not found in HPF; variants of these language extensions are found in Fortran D and Vienna Fortran. The first feature is the **ON** clause; the **ON** clause allows users to specify which processor is to execute each iteration of a loop. The second feature is an intrinsic function that can be used to carry out reduction in a parallel (forall) loop.

In irregular problems, data access patterns and workload are usually known only at runtime, hence decisions regarding data and work distributions are made at runtime. These on-the-fly decisions require special runtime support. A set of procedures have been developed, called *CHAOS*, that can be used by an HPF style compiler. CHAOS is a successor of *PARTI* [15] and provides support for managing user-defined distributions, partitioning loop iterations, remapping data and index arrays, and generating optimized communication schedules.

The methods proposed in this paper are implemented in the Syracuse Fortran 90D prototype compiler. Templates from real application codes are employed to study performance. The compiler generates parallel codes for irregular problems by embedding CHAOS runtime procedures.

Examples of targeted applications are introduced in Section 2. Section 3 presents the functionality of the runtime support and an overview of the existing data parallel languages. Section 4 describes the language support for irregular distributions. Performance results of the runtime system for templates from real-applications are presented in Section 5. The templates are derived from the application codes which will be described in Section app-codes. The performance of the compiler-generated codes is compared to that of hand-written codes. Section 6 presents conclusions.

## 2 Application Codes

It is useful to describe application codes to introduce the motivation behind irregular distributions. The loop structures of two application codes (an unstructured Euler solver and a molecular dynamics code) are described in this section. They consist of a sequence of loops with indirectly accessed arrays and are similar to those depicted in Figure 1.

The first application code is an unstructured Euler solver used to study the flow of air over an airfoil [11]. Complex aerodynamic shapes require high resolution meshes and, consequently, large numbers of mesh points. Physical values (e.g. velocity, pressure) are associated with each mesh vertex. These values are called *flow variables* and are stored in arrays. These arrays are called *data arrays*. Calculations are carried out using loops over the list of edges that define the connectivity of the vertices.

To parallelize an unstructured Euler solver, mesh vertices must be partitioned (i.e. arrays that store flow variables). Since meshes are typically associated with physical objects, a spatial location can often be associated with each mesh point. The spatial locations of the mesh points and the connectivity of the vertices is determined by the mesh generation strategy [11]. Figure 3 depicts a mesh generated by such a process. This is an unstructured mesh representation of a three dimensional aircraft wing.

The way in which the vertices of such irregular computational meshes are numbered frequently does not have a useful correspondence to the connectivity pattern (edges) of the mesh. During mesh generation, vertices are added progressively to refine the mesh. While new vertices are added, new edges are created or older ones are moved around to fulfill certain mesh generation criteria. This

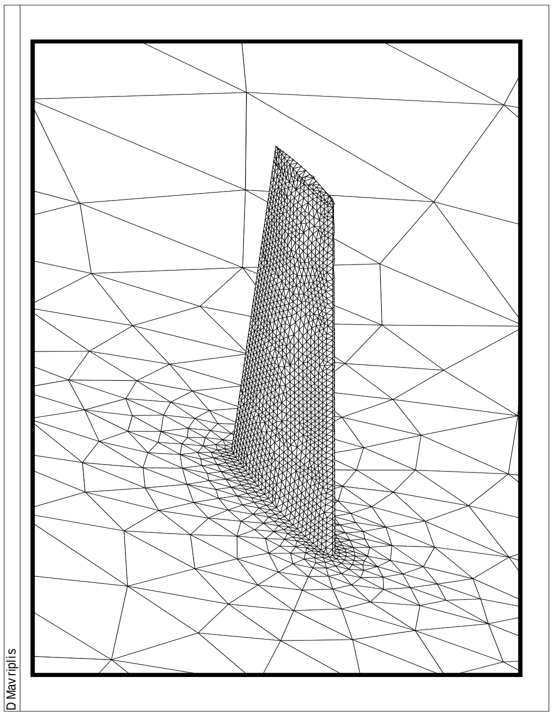



Figure 3: An Unstructured Mesh

causes the apparent lack of correspondence between the vertex numbering and edge numbering. One way to solve this problem is to renumber the mesh completely after the mesh has been generated.

Mesh points are partitioned to minimize communication. Recently, promising heuristics have been developed that can use one or several of the following types of information: 1) spatial locations of mesh vertices, 2) connectivity of the vertices, and 3) estimate of the computational load associated with each mesh point. For instance, a user might choose a partitioner that is based on coordinates [1]. A coordinate bisection partitioner decomposes data using the spatial locations of vertices in the mesh. If the user chooses a graph-based partitioner, the connectivity of the mesh could be used to decompose the mesh.

The next step in parallelizing this application involves assigning equal amounts of work to processors. A Euler solver consists of a sequence of loops that sweep over a mesh. Computational work associated with each loop must be partitioned among processors to balance load. Consider a loop that sweeps over mesh edges, closely resembling the loop depicted in Figure 1. Mesh edges are partitioned so that 1) load balance is maintained, and 2) computations mostly employ locally stored data.

Other unstructured problems have similar indirectly accessed arrays. For instance, consider the non-bonded force calculation in the molecular dynamics code, CHARMM [3], shown in Figure 4. Force components associated with each atom are stored as Fortran arrays. The loop L1 sweeps over all atoms. In this discussion, it is assumed that L1 is a parallel loop while L2 is a sequential one. The

```
L1: do i = 1, NATOM
     L2: do index = 1, INB(i)
              j = Partners(i, index)
              Calculate dF (x, y and z components).
              Subtract dF from F_j.
              Add dF to F_i
         end do
    end do
```

Figure 4: Non-bonded Force Calculation Loop from CHARMM

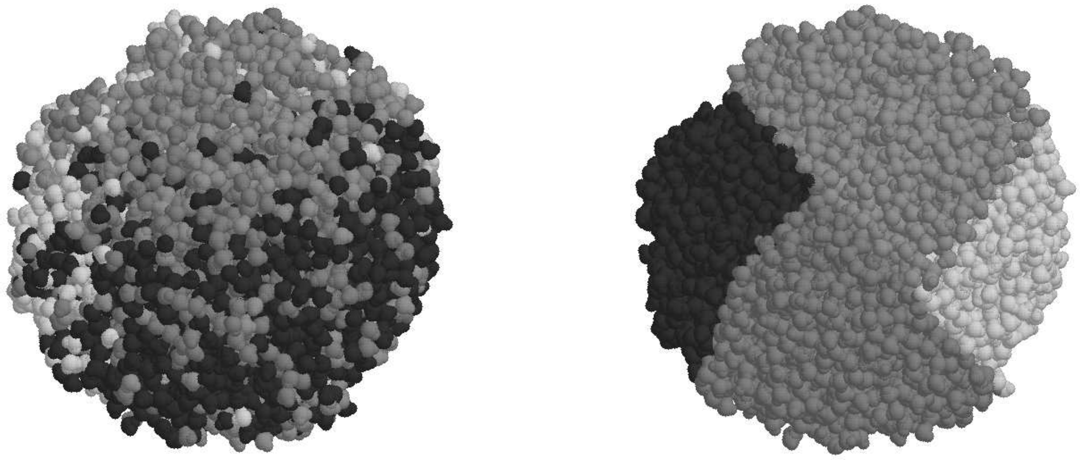
loop iterations of L1 are distributed over processors. All computation pertaining to iteration $i$ of L1 is carried out on a single processor, so loop L2 need not be parallelized.

It is assumed that all atoms within a given cutoff radius interact with each other. The array **Partners(i, *)** lists all the atoms that interact with atom $i$. The inner loop calculates the three force components (x, y, z) between atom $i$ and atom $j$ (van der Waal's and electrostatic forces). They are then added to the forces associated with atom $i$ and subtracted from the forces associated with the atom $j$.

The force array elements are partitioned in a way as to reduce interprocessor communication in the non-bonded force calculation loop (Figure 4). Figure 5 depicts two possible distributions of atoms of a Myoglobin and 3830 water molecules onto eight processors. Shading is used to represent the assignment of atoms to processors. Data sets associated with the sequential version of CHARMM assign each atom an index number which does not reflect locality. Figure 5(a) depicts a distribution that assigns consecutively numbered sets of atoms to each processor, i.e. a `BLOCK` distribution. Since nearby atoms interact, the choice of a `BLOCK` distribution is likely to result in a large volume of communication. Consider, instead, a distribution based on the spatial locations of atoms. Figure 5(b) depicts a distribution of atoms to processors carried out using an inertial bisection partitioner. Compare Figure 5(a) and 5(b), the later figure has a much smaller amount of surface area between portions of the molecules associated with each processor.

## 3 Runtime Support

This section is an overview of the principles and functionality of the CHAOS runtime support library, a superset of the PARTI library [15].



(a) BLOCK Distribution (b) Irregular Distribution

Figure 5: Distribution of Atoms on 8 Processors

### 3.1 Overview of CHAOS

The CHAOS runtime library has been developed to efficiently handle problems that consist of a sequence of clearly demarcated concurrent computational phases. Solving such concurrent irregular problems on distributed memory machines using CHAOS runtime support involves six major steps (Figure 6). The first four steps concern mapping data and computations onto processors. The next two steps concern analyzing data access patterns in a loop and generating optimized communication calls. A brief description of these phases follows.

Initially, arrays are decomposed into either regular or irregular distributions.

A. **Data Distribution** : Phase A calculates how data arrays are to be partitioned by making use of partitioners provided by CHAOS or by the user. CHAOS supports a number of parallel partitioners that use heuristics based on spatial positions, computational load, connectivity, etc. The partitioners return an irregular assignment of array elements to processors; this is stored as a CHAOS construct called the *translation table*. A translation table is a globally accessible data structure which lists the home processor and offset address of each data array element. The translation table may be replicated, distributed regularly, or stored in a paged fashion, depending on storage requirements.

B. **Data Remapping** : Phase B remaps data arrays from the current distribution to the newly calculated irregular distribution. A CHAOS procedure `remap` is used to generate an optimized

| | | |
|---|---|---|
| Phase A : | Data Partitioning | Assign elements of data arrays to processors |
| Phase B : | Data Remapping | Redistribute data array elements |
| Phase C : | Iteration Partitioning | Allocate iterations to processors |
| Phase D : | Iteration Remapping | Redistribute indirection array elements |
| Phase E : | Inspector | Translate indices; Generate schedules |
| Phase F : | Executor | Use schedules for data transportation;<br>Perform computation |

Figure 6: Solving Irregular Problems

*communication schedule* for moving data array elements from their original distribution to the new distribution.

C. **Loop Iteration Partitioning** : Phase C determines how loop iterations should be partitioned across processors. There are a large number of possible schemes for assigning loop iterations to processors based on optimizing load balance and communication volume. CHAOS uses the *almost-owner-computes* rule to assign loop iterations to processors. Each iteration is assigned to the processor which owns a majority of data array elements accessed in that iteration. This heuristic is biased towards reducing communication costs.

D. **Remapping Loop Iterations** : Phase D is similar to phase B. Indirection array elements are remapped to conform with the loop iteration partitioning. For example, in Figure 1, once loop L2 is partitioned, indirection array elements $edge1(i)$ and $edge2(i)$ used in iteration $i$ are moved to the processor which executes that iteration.

E. **Inspector** : Phase E carries out the preprocessing needed for communication optimizations and *index translation.*

F. **Executor** : Phase F uses information from the earlier phases to carry out computation and communication. Communication is carried out by CHAOS data transportation primitives which use communication schedules constructed in Phase E.

Phase F is typically executed many times in real application codes, however, phases A through E are executed only once if the data access patterns do not change. When programs change data access patterns but maintain good load balance, phases E and F are repeated. If programs require remapping of data arrays from the current distribution to a new distribution, all phases are executed again.

A brief presentation of some important CHAOS features that are useful to parallelize irregular programs is given in the following sections.

### 3.2 Data Access Descriptors – Translation Tables

When an array is irregularly distributed, a mechanism is needed to retrieve required elements of that array. CHAOS supports a translation mechanism using a data structure called the *translation table*. A translation table lists the home processor and the local address in the home processor's memory for each element of the irregularly distributed array. In order to access an element $A(m)$ of distributed array $A$, a translation table lookup is necessary to find out the location of $A(m)$.

The data structure translation table has the following fields:

1. global size N,
2. distribution type T,
3. block size B,
4. local size L,
5. processor list $\vec{p}$, and
6. offset list $\vec{l}$.

The first four fields are used to represent regular distributions such as BLOCK and CYCLIC. These fields are not enough to represent irregular distributions. Two additional fields, processor list and offset list, are used in this case. The processor list $\vec{p}$ gives the home processor of each array element; offset list $\vec{l}$ gives the local addresses of the elements. A translation table lookup, which is aimed at computing the home processor and the offset associated with a global distributed array index, is known as a *dereference request*. Any preprocessing aimed at communication optimizations needs to perform dereferencing, since it is required to determine where elements reside.

Several considerations arise during the design of data structures for a translation table. Depending on the specific parameters of the problem, there is usually a trade-off involving storage requirements, table lookup latency and table update costs. Of these, table lookup costs are of primary consideration in adaptive problems, since preprocessing must be repeated frequently, and must be efficient.

The fastest table lookup is achieved by replicating the translation table in each processor's local memory. This type of translation table is a *replicated translation table*. Clearly, the storage cost for this type of translation table is $O(NP)$, where $P$ is the number of processors and $N$ is the array size. However, the dereference cost in each processor is constant and independent of the number of processors involved in the computation. Note that, for the replicated translation table, the translation table in each processor is identical.

Due to memory considerations, it is not always feasible to place a copy of the translation table on each processor. The approach taken in these cases is to distribute the translation table between processors. This type of translation table is a *distributed translation table.* Earlier versions of PARTI supported a translation table that was distributed between processors in a blocked fashion. This is accomplished by distributing the translation table by blocks, i.e., putting the first $N/P$ elements on the first processor, the second $N/P$ elements on the second processor, etc. When an element $A(m)$ of the distributed array $A$ is accessed, the home processor and local offset are found in the portion of the distributed translation table stored in processor $\lfloor((m-1)/N)\times P\rfloor+1$. Distributed translation tables have the highest utilization of available distributed memory for a fixed-size irregularly-distributed array. The dereference requests on the other hand, now may require a communication step, since some portions of the translation table may not be residing in the local memory. Similarly, table reorganization also requires interprocessor communication since each processor is authorized to modify only a limited portion of the translation table.

Besides supporting replicated and distributed translation tables, CHAOS also supports an intermediate degree of replication with *paged translation tables.* In this scheme, the translation table is divided into pages, and pages are distributed across processors. Processors that refer to a page frequently receive a copy of the page, making subsequent references local. A more detailed description of this scheme is presented in Das et al. [5].

Figure 7 depicts the three translation table structures of a graph partitioned over 2 processors. Only the processor list $\vec{p}$ and offset list $\vec{l}$ are displayed. The numbers above arrays are the index numbers of nodes. Figure 7(a) shows an irregular distributed. Nodes 1, 2, and 5 are assigned to processor $P0$, and nodes 3, 4, and 6 to processor $P1$. The distributed translation table shown in Figure 7(b) assigns first three elements of the lists $\vec{p}$ and $\vec{l}$ on $P0$ and the last three to $P1$. By contrast, the replicated translation table replicates all the 6 elements of $\vec{p}$ and $\vec{l}$ on both processors, as shown in Figure 7c. Figure 7d illustrate the structure of a paged translation table with the page size of 2. Each processor owns two pages. The dashed page on $P0$ is copied from $P1$ as the result of remote references of node 5 from $P0$ to $P1$.

### 3.3 Data Redistribution

For efficiency reasons, in scientific programs, distribution of data arrays may have to be changed between computational domains or phases. For instance, as computation progresses in an adaptive problem, the work load and distributed array access patterns may change based on the nature of problem. This change might result in a poor load balance among processors. Hence, data must be redistributed periodically to maintain balance.

To obtain an irregular data distribution for an irregular concurrent problem, data arrays are distributed in a known distribution, $\delta_A$. Then, a heuristic method is applied to obtain an irregular distribution $\delta_B$. Once the new data distribution is obtained, all data arrays associated with distribution

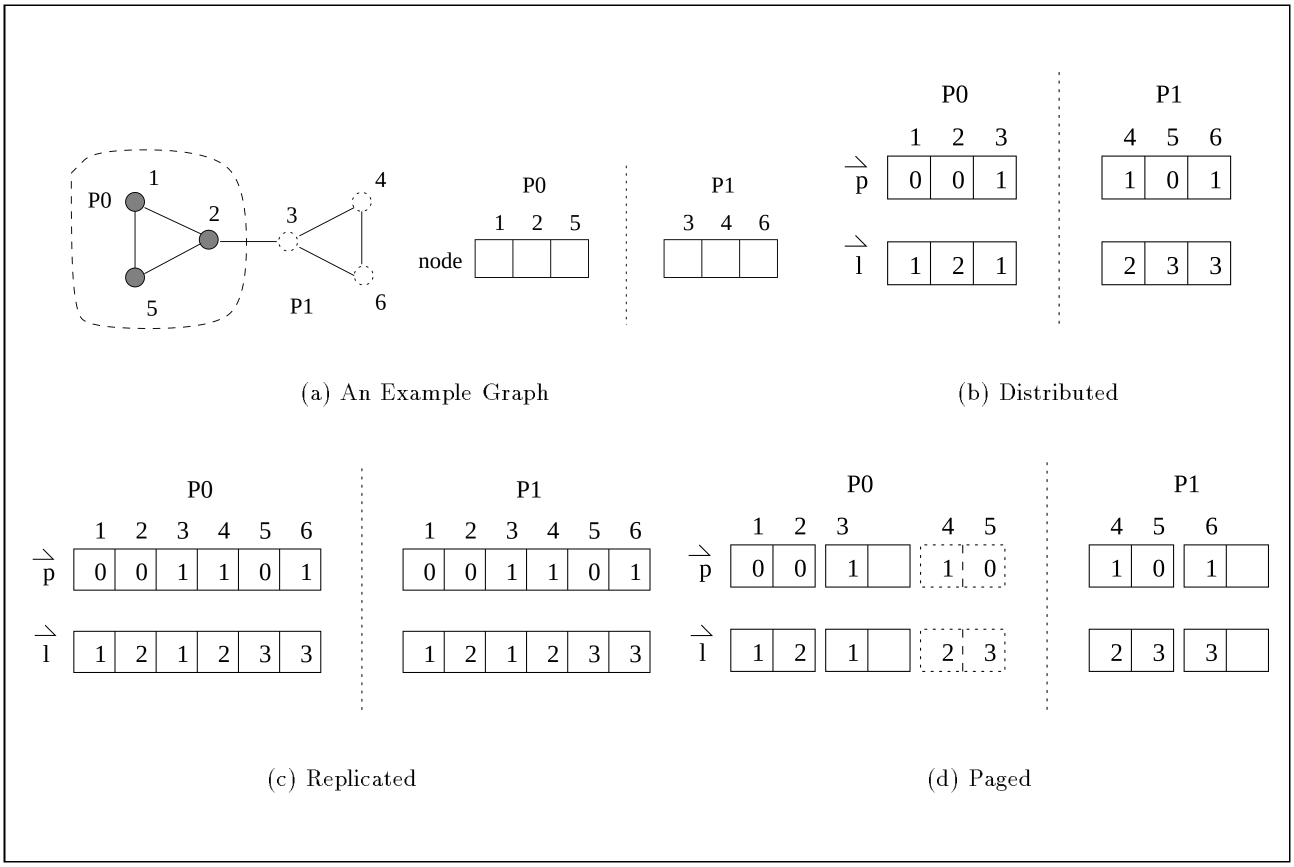



Figure 7: Translation Table

$\delta_A$ must be transformed to distribution $\delta_B$. Similarly, the loop iterations and the indirection arrays associated with the loop must be remapped.

To redistribute data and loop iterations, a runtime procedure called *remap* has been developed. This procedure takes as input the original and the new distribution in the form of translation tables and returns a communication schedule. This schedule can be used to move data between initial and subsequent distributions.

## 3.4 Loop Iteration Partitioning

Once data arrays are partitioned, loop iterations must also be partitioned. Loop partitioning refers to determining which processor will evaluate which expressions of the loop body. Loop partitioning can be performed at several levels of granularity. At the finest level, each operation may be individually assigned to a processor. At the coarsest level, a block of iterations may be assigned to a processor, without considering the data distribution and access patterns. Both approaches are expensive. In the first case, the amount of preprocessing overhead can be very high, and in the second case, communication cost can be very high. The approach used by CHAOS represents a compromise. Each loop iteration is individually considered prior to processor assignment.

To partition loop iterations, a set of runtime procedures has been developed. These procedures compute, with the current known distribution of loop iterations, a list containing the home processors of the distinct data references for each local iteration. Currently, the heuristic used for iteration partitioning is the "almost owner computes" rule, in which an iteration is assigned to the processor which owns the majority of the elements participating in that particular iteration.

Following the loop iteration distribution, the data references in each iteration must be remapped to conform with the new loop iteration distribution. An inspector phase is carried out to remap data references. A communication schedule is built in the inspector phase and it is used to gather the new data references.

### 3.5 Communication Schedules

As described in Section 3.1, a communication schedule is used to fetch off-processor elements into a local buffer and to scatter these elements back to their home processors after the computational phase is over. Communication schedules determine the number of communication startups and the volume of communication, so it is important to optimize them.

The schedule for processor $p$ stores the following information:

1. send list – a list of arrays that specifies the local elements of a processor $p$ required by all processors,

2. permutation list – an array that specifies the data placement order of off-processor elements in the local buffer of processor $p$,

3. send size – an array that specifies sizes of out-going messages of processor $p$ to all processors, and

4. fetch size – an array that specifies sizes of in-coming messages of processor $p$ from all processors.

### 3.6 Data Transportation

While communication schedules store data send/receive patterns, the CHAOS data transportation procedures actually move data using these schedules. The procedure *gather* can be used to fetch a copy of off-processor elements. The procedure *scatter* can be used to send off-processor elements.

## 4 Language Support

A wide range of languages, such as Vienna Fortran [4], pC++ [7], Fortran D [6] and HPF [10], provide a rich set of directives that allow users to specify desired data decompositions. With these directives, compilers can partition loop iterations and generate the communication required to parallelize the code. This research is presented in the Fortran D context. However, the same could be extended for

```
S1          REAL A(N, N)
S2    C$    DECOMPOSITION D(N, N)
S3    C$    ALIGN A(I, J) with D(I, J)
S4    C$    DISTRIBUTE D(*, BLOCK)
```

Figure 8: Fortran D Data Distribution Specifications

other languages. The following discussion involves existing Fortran D language support and compiler performance for irregular problems.

Fortran D provides users with explicit control over data partitioning using **DECOMPOSITION**, **ALIGN** and **DISTRIBUTE** directives. In Fortran D a template, called a *distribution*, is declared and used to characterize the significant attributes of a distributed array. The distribution fixes the size, dimension, and way in which the array is to be partitioned between processors. A distribution is produced using two declarations. The first declaration is **DECOMPOSITION**. Decomposition fixes the name, dimensionality and size of the distributed array template. The second declaration is **DISTRIBUTE**. DISTRIBUTE is an executable statement and specifies how a template is to be mapped onto the processors. Fortran D provides the user with a choice of several regular distributions. In addition, a user can explicitly specify how a distribution is to be mapped onto the processors. A specific array is associated with a distribution using the Fortran D statement **ALIGN**.

In the example shown in Figure 8, $D$ is declared to be a two-dimensional decomposition of size $N \times N$. Array $A$ is then aligned with the decomposition $D$. Distributing decomposition $D$ by `(*,BLOCK)` results in a column partition of arrays aligned with $D$. A detailed description of the language can be found in Fox, et al. [6]. The data distribution specifications are then treated as comment statements in a sequential machine Fortran compiler. Hence, a program written with distribution specifications can be compiled and executed on a sequential machine.

## 4.1 Support for Irregular Distributions

Fortran D supports irregular data distributions and dynamic data decomposition, i.e. changing the alignment or distribution of a decomposition at any point in the program. In Fortran D an irregular partition of distributed array elements can be *explicitly* specified. Figure 9 depicts an example of such a Fortran D declaration. In statement S3 of Figure 9, two 1D decompositions, each of size `N`, are defined. In statement S4, decomposition `reg` is partitioned into equal sized blocks, with one block assigned to each processor. In statement S5, array `map` is aligned with distribution `reg`. Array `map` will be used to specify (in statement S7) how distribution `irreg` is to be partitioned between processors. An irregular distribution is specified using an integer array; when $map(i)$ is set equal to $p$, element $i$ of the distribution `irreg` is assigned to processor $p$. A data partitioner can be invoked to set the values of the permutation array. Support for irregular distributions has been provided by Vienna Fortran also [4].

```
S1        REAL*8 x(N),y(N)
S2        INTEGER map(N)
S3   C$   DECOMPOSITION reg(N),irreg(N)
S4   C$   DISTRIBUTE reg(block)
S5   C$   ALIGN map with reg
S6        ... set values of map array using some mapping method ..
S7   C$   DISTRIBUTE irreg(map)
S8   C$   ALIGN x,y with irreg
```

Figure 9: Fortran D Irregular Distribution

```
C    Sweep over edges: Loop L2
     FORALL (i = 1: nedge)
S1     REDUCE (SUM, y(edge1(i)), f(x(edge1(i)), x(edge2(i))))
S2     REDUCE (SUM, y(edge2(i)), g(x(edge1(i)), x(edge2(i))))
     END FORALL
```

Figure 10: Example Irregular Loop in Fortran D

### 4.2 Computational Loop Structures

Figure 10 shows an irregular Fortran 90D FORALL loop that is equivalent to the sequential loop L2 in Figure 1. Loop L2 represents a sweep over the edges of an unstructured mesh. Since the mesh is unstructured, an indirection array must be used to access the vertices during a loop over the edges. In loop L2, a sweep is carried out over the edges of the mesh and the reference pattern is specified by integer arrays `edge1` and `edge2`. Loop L2 carries out reduction operations which are the only types of dependency between different iterations of the loop in which they may produce a value to be accumulated (using an associative and commutative operation) in the same array element. Figure 3 shows an example of an unstructured mesh over which such computations will be executed. For example, loop L2 represents a sweep over the edges of a mesh in which each mesh vertex is updated using the corresponding values of its neighbors (directly connected through edges). Each vertex of the mesh is updated as many times as the number of neighboring vertices.

The implementation of the FORALL construct in Fortran D follows copy-in-copy-out semantics; loop carried dependencies are not defined. In the present implementation, loop carried dependencies that arise due to reduction operations are allowed. The reduction operations are specified in a FORALL construct using the Fortran D REDUCE construct. Reduction inside a FORALL construct is important for representing computations such as those found in sparse and unstructured problems. This representation also preserves explicit parallelism available in the underlying computations.

### 4.3 Loop Iteration Distribution

Once data arrays are partitioned, computational work must also be partitioned. One convention is to compute a program assignment statement S in the processor that owns the distributed array element on S's left hand side. This convention is normally referred to as the *owner computes* rule. If the left hand side of S references a replicated variable then the work is carried out in all processors. One drawback to the owner computes rule in sparse codes is that communication might be required within loops, even in the absence of loop-carried dependencies. For example, consider the following loop:

```
    FORALL i = 1, N
S1  x(ib(i)) = ......
S2  y(ia(i)) = x(ib(i))
    END FORALL
```

This loop has a loop independent dependence between S1 and S2, but no loop carried dependencies. If work is assigned using the owner computes rule, for iteration $i$, statement S1 would be computed on the owner of x(ib(i)), OWNER(x(ib(i))), while statement S2 would be computed on the owner of y(ia(i)), OWNER(y(ia(i))). The value of y(ib(i)) would have to be communicated whenever OWNER(x(ib(i))) $\neq$ OWNER(y(ia(i))).

In Fortran D and Vienna Fortran, a user can specify on which processor to carry out a loop iteration using the **ON** clause. For example, in Fortran D, a loop could be written as

```
    FORALL i = 1,N on HOME(x(i))
S1  x(ib(i)) = ......
S2  y(ia(i)) = x(ib(i))
    END FORALL
```

This means that iteration $i$ must be computed on the processor on which $x(i)$ resides, where the sizes of arrays `ia` and `ib` are equal to the number of iterations. A similar HPF directive EXECUTE-ON-HOME, proposed in the journal of development [9], provides such a capability.

A method proposed by Ponnusamy et al. [13] employs a scheme that executes a loop iteration on the processor that is the home of the largest number of distributed array references in that iteration. This is referred to as the *almost owner computes* rule.

```
C$  EXECUTE (i) ON_HOME(map(i))
    FORALL i = 1,N
S1  x(ib(i)) = ......
```

```
C     Initially arrays are distributed in blocks
C$    DECOMPOSITION reg(14026)
C$    DISTRIBUTE reg(BLOCK)
C$    ALIGN x, y, dx, dy WITH reg
      ...
S1    Obtain new distribution format (map) from the extrinsic partitioner
C$    DISTRIBUTE reg (map)
      ...
C     Calculate DX and DY
C$    EXECUTE (i,*) ON_HOME(reg(i))
      FORALL (i = 1: natom)
        FORALL (j = inblo(i): inblo(i+1) - 1)
          REDUCE (SUM, dx(jnb(j)), x(jnb(j)) - x(i))
          REDUCE (SUM, dy(jnb(j)), y(jnb(j)) - y(i))
          REDUCE (SUM, dx(i), x(i) - x(jnb(j)))
          REDUCE (SUM, dy(i), y(i) - y(jnb(j)))
        END FORALL
      END FORALL
```

Figure 11: Non-bonded Force Calculation Loop of CHARMM

S2 y(ia(i)) = x(ib(i))

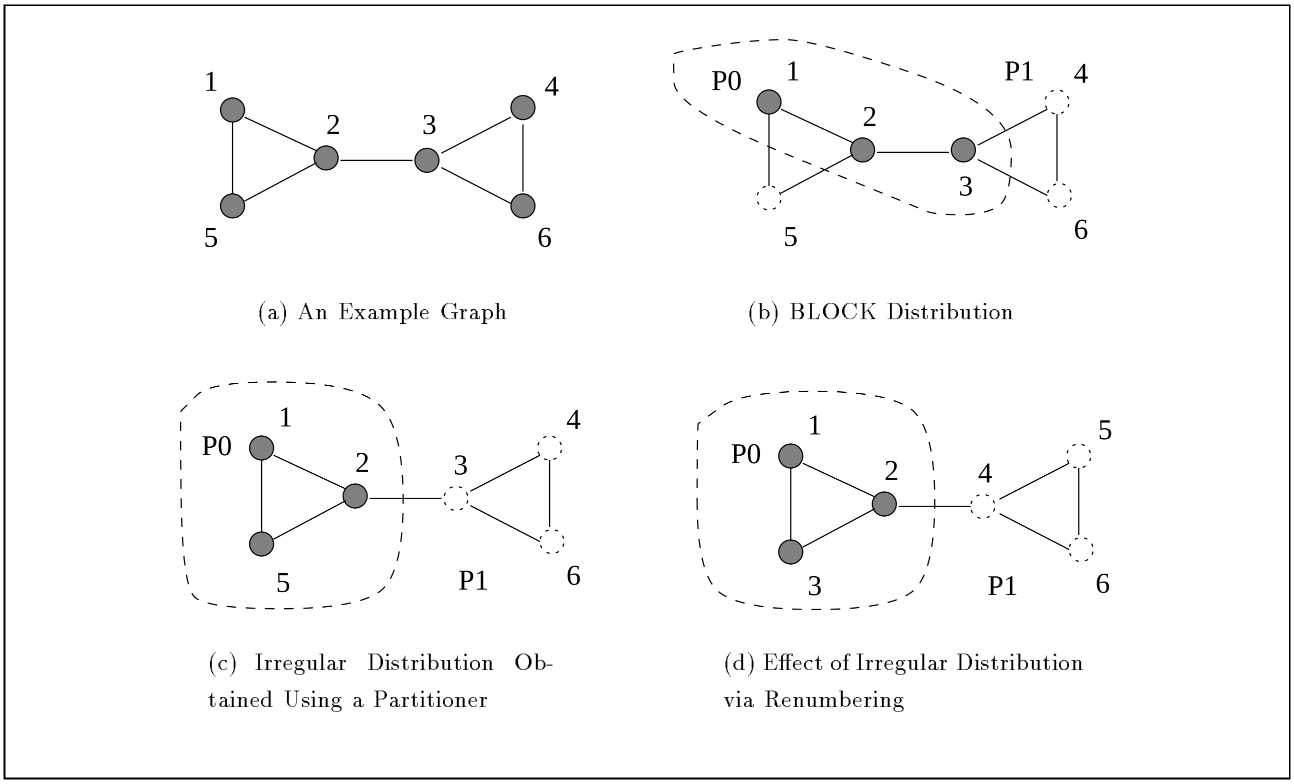
**END FORALL**

In the above example, the *proposed* HPF directive EXECUTE-ON-HOME has been used to support the almost owner computes rule. In this example, an iteration $i$ is assigned to the processor $map(i)$. A user-defined function determines the values of array `map`. This function assigns an iteration to the processor which owns the majority of the distributed array elements referenced in that iteration. Figure 11 depicts an irregular loop from CHARMM in Fortran 90D with the HPF EXECUTE-ON-HOME directive for partitioning loop iterations. The inner loop iterations are executed on processors which own $reg(i)$, where `reg` is the decomposition to which arrays `x`, `y`, `dx`, and `dy` are aligned. The array `inblo` is replicated on all processors.

## 4.4 Applications to HPF

Thus far, the runtime support for irregular problems has been presented in the context of the Fortran D system, these methods can be used in HPF compilers as well.

The current version of HPF does not support non-standard distributions. However, HPF can indirectly support such distributions by reordering array elements in ways that lead to reduced communication requirements. Applications scientists have frequently employed variants of this approach when porting irregular codes to parallel architectures [17]. A partitioner is first used to obtain a




(a) An Example Graph

(b) BLOCK Distribution

(c) Irregular Distribution Obtained Using a Partitioner

(d) Effect of Irregular Distribution via Renumbering

Figure 12: An Example of Renumbering Technique

mapping of array elements to processors. Array elements are then reordered so that elements mapped to a given processor are assigned to consecutive locations. When the same number of elements are mapped to each processor, and the number of processors evenly divides the array size, the benefits of an irregular distribution can immediately be obtained using a BLOCK distributed reordered array. The necessary preprocessing can be carried out in HPF by calling extrinsic procedures that invoke partitioners to obtain array mappings along with extrinsic procedures that reorder data and indirection arrays. The following example illustrates the reordering technique.

Figure 12 depicts a simple graph, an irregular grid with 6 nodes and 7 edges, partitioned between two processors. The graph can be described based on the simple Fortran D program shown in Figure 10. The graph is used to describe the flow of data between elements of arrays `x` and `y`; an edge between nodes $n_1$ and $n_2$ means the value of $x(n_1)$ is accumulated to $y(n_2)$ and the value of $x(n_2)$ is accumulated to $y(n_1)$.

In the example shown in Figure 12, it is clear partitioning should occur to (1) allocate the same number of nodes to processors, and (2) minimize the number of cross-edges existing between processors, i.e., minimize the number of edges for which both end-nodes do not lie on the same processor. Figure 12(a) shows the original graph. In Figure 12(b) the graph is partitioned in BLOCK format based on node numbers. Nodes 1, 2, and 3 are assigned to processor 0 and the rest to processor 1. The cross-edges in this distribution are (1, 5), (2, 5), (3, 6), and (3, 4). Figure 12(c) shows a better

```
S1   INTERFACE
S2   EXTRINSIC(HPF_LOCAL) SUBROUTINE binary_dissection_2D(reorder, x, y, n)
S3   REAL*8, DIMENSION(:), INTENT(IN) :: x, y
S3   INTEGER INTENT(IN) :: n
S4   INTEGER, DIMENSION(:), INTENT(OUT) :: reorder
S7   END SUBROUTINE binary_dissection_2D
S8   END INTERFACE
```

Figure 13: Interfacing an Extrinsic Partitioner Procedure

```
!HPF$   TEMPLATE reg(N), reg1(M)
!HPF$   DISTRIBUTE(BLOCK) ONTO P :: reg, reg1
!HPF$   ALIGN WITH reg :: x, y, reorder
!HPF$   ALIGN WITH reg1 :: edge1, edge2, temp
        ...
C       use an extrinsic partitioner procedure to obtain reorder array
        CALL binary_dissection_2D(reorder, x, y, n_local)
C       use an extrinsic procedure to reorder data arrays
        CALL renumber_data_array(reorder, x, n_local)
        CALL renumber_data_array(reorder, y, n_local)
C       use an extrinsic procedure to renumber indirection arrays
        CALL renumber_indirection_array(reorder, edge1, n_localedge)
        CALL renumber_indirection_array(reorder, edge2, n_localedge)
        ...
C       Sweep over edges: Loop L2
        FORALL(i=1:nedge) temp(i) = f(x(edge1(i)),x(edge2(i)))
        y = SUM_SCATTER(temp, y, edge1)
        FORALL(i=1:nedge) temp(i) = g(x(edge1(i)),x(edge2(i)))
        y = SUM_SCATTER(temp, y, edge2)
```

Figure 14: Irregular Distribution and Loops in HPF

distribution of the same graph, with a smaller number of cross-edges. In this distribution, nodes 1, 2, and 5 are assigned to processor 0 and the rest to processor 1; there is only one cross-edge, edge (2, 3). This distribution results in an arbitrary assignment of nodes to processors, or irregular distribution of nodes. The effect of this distribution can be obtained by assigning new indices to the nodes so that contiguously numbered nodes are assigned to each processor. When this renumbering is carried out, the graph depicted in Figure 12(c) is transformed to the graph shown in Figure 12(d). Figure 12(c) and 12(d) depict identical graph partitions; the two figures label nodes (and consequently edges) with different numbers. Thus, in Figure 12(d), the cross-edge is edge (2, 4). Note that Figure 12(c) corresponds to an irregular distribution in a data parallel language, whereas Figure 12(d) corresponds to a BLOCK distribution.

A non-HPF procedure can be interfaced with HPF programs using the EXTRINSIC directive, as shown in Figure 13. In statement S2, the interface from HPF to a partitioner *binary_dissection_2D* is specified. The directive HPF_LOCAL indicates that the procedure *binary_dissection_2D* has been written in local HPF style. This particular procedure uses information provided in arrays `x` and `y` and writes the result of the partitioning to the permutation array `reorder`. The statements S3 and S4 specify the input (`x` and `y`) and output (`reorder`) parameters.

Figure 14 illustrates the reordering technique expressed in HPF. To begin, arrays `x`, `y`, and `reorder` are distributed by BLOCK. Next, an extrinsic partitioner procedure is called to determine the values of array `reorder`. An extrinsic procedure, *renumber_data_array*, is invoked to reorder data arrays `x` and `y` based on the values of array `reorder`. After the reordering is completed, the $i$th element of `x` array is moved to the position $reorder(i)$ and another extrinsic function *renumber_indirection_array* is called to update arrays `edge1` and `edge2` so that values of these arrays reflect the new positions of array elements of `x` and `y`, i.e., the value of $edge1(i)$ is modified to $reorder(edge1(i))$.

The current version of HPF does not support the REDUCE construct that is provided by Fortran D. However, the functionality of the type of irregular loop shown in Figure 1 can be expressed in HPF with the help of intrinsic procedures. Figure 14 depicts a method of expressing the irregular loop L2 in HPF. Here, the HPF intrinsic function *SUM_SCATTER* is used to express an array combining operation. A statement in a sequential irregular loop, which has indirectly accessed arrays on both right and left hand sides of the statement, can be written in HPF as two separate phases: (1) a FORALL loop to carry out the computation in the right hand side and store the values to a temporary array `temp`, and (2) an intrinsic function *SUM_SCATTER* to scatter and combine the elements of array `temp` to array `y`.

Although Figure 14 shows that the irregular loop in Figure 1 can be expressed in HPF, additional preprocessing operations must be performed. The reordering technique has to be used because HPF does not support irregular distribution. The temporary array *temp* has to be introduced for FORALL statements because HPF does not provide the REDUCE construct. Both features, irregular distributions and REDUCE constructs supported by Fortran D, provide users with proper facilities to specify appropriate distributions for applications and to express the reduction operations for irregular loops.

The preceeding discussion assumes that (1) the number of array elements can be evenly divided by the number of processors and (2) the same number of elements are assigned to each processor. In many cases it may be advantageous to assign different numbers of data elements to processors in order to balance the workload. To accomplish this, first the user declares the original array as an oversized array (in BLOCK distribution); next, a partitioner is called to reassign the array elements to processors such that no more than a given number of elements are assigned to any processor.

Assume that a one-dimensional array `A` has $N \times P$ elements, where $N$ is the number of elements on each processor and $P$ is the number of processors. The user decides that no more than $M$ ($M > N$) array elements may be assigned to any processor.

1. The user declares `A` as a $M \times P$ BLOCK distributed array. Originally, only the first $N \times P$ elements of `A` will be initialized with meaningful values, and the last $(M - N) \times P$ elements of

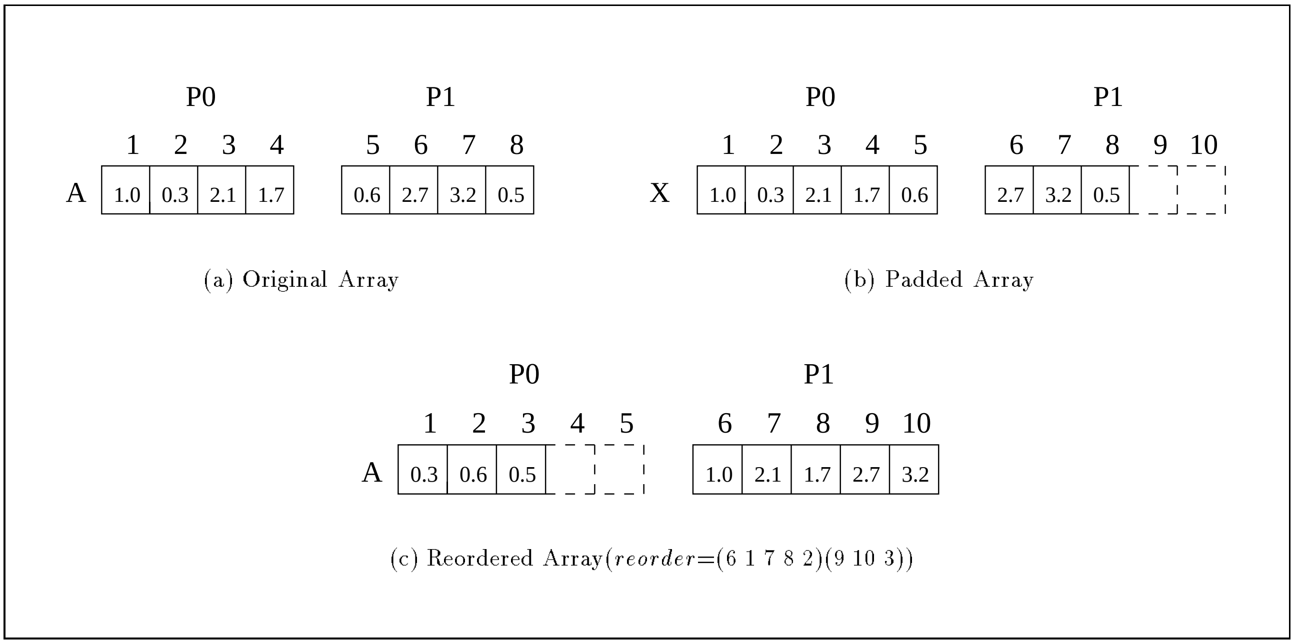



Figure 15: An Example of Array Padding and Reordering

A are unused storage.

2. The user then employs a partitioner that is constrained to assign no more than $M$ elements to each processor, where $M > N$. The partitioner returns a reordering array `reorder` which maps $A(i)$ to $A(reorder(i))$, $1 \leq i \leq N \times P$. In order to assign a $A(i)$ to processor $p$, where $0 \leq p < P$, the partitioner defines $reorder(i)$ as $M \times (p-1) < reorder(i) \leq M \times p$.

The reordering array `reorder` can then be used to reorder the elements of `A`. Once the reordering is complete, the reordered array `A` will still have $(M - N) \times P$ elements that will not contain meaningful values; these *ghost* elements will now be scattered throughout the array.

Figure 15 presents an example of array reordering. An array with 8 meaningful elements, Figure 15(a), is declared as a 10 element BLOCK array, as shown in Figure 15(b). Figure 15(c) depicts the result of carrying out a reordering based on the `reorder` array returned by a partitioner. The $i$th element of `A` is moved to position $reorder(i)$, e.g., when $reorder(1) = 6$, $A(1)$ in Figure 15(b) is moved to $A(6)$ in Figure 15(c). Notice that there are two ghost elements (in dashed lines) at the middle of the reordered array.

# 5 Compiler Support and Experimental Results

This section presents the compiler transformations used to handle irregular templates that appear in the molecular dynamics code, CHARMM [3], and fluid dynamics code, EUL3D [11]. Runtime support has been incorporated in the Fortran 90D compiler being developed at Syracuse University [2]. The

Table 1: Effect of Data Distribution – Hand-Coded – 32 Processors

| (Time in Secs) | Coordinate Bisection | | Block Partition | |
|---|---|---|---|---|
| | 53K Mesh | 14K Atoms | 53K Mesh | 14K Atoms |
| Partitioner | 2.4 | 0.7 | 0.0 | 0.0 |
| Remap | 2.6 | 2.5 | 1.6 | 0.0 |
| Inspector | 0.9 | 0.7 | 0.5 | 1.4 |
| Executor | 14.1 | 93.5 | 34.6 | 187.9 |
| Total | 20.0 | 97.4 | 36.7 | 189.3 |

Table 2: Performance for Block Distribution – EUL3D Loop

| Tasks (Time in Secs) | Hand | | | | Compiler | | | |
|---|---|---|---|---|---|---|---|---|
| | 10K Mesh | | 53k Mesh | | 10K Mesh | | 53k Mesh | |
| | Procs | | Procs | | Procs | | Procs | |
| | 8 | 16 | 32 | 64 | 8 | 16 | 32 | 64 |
| Partitioner | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| Remap | 0.9 | 0.4 | 1.6 | 1.0 | 0.9 | 0.5 | 1.6 | 1.0 |
| Inspector | 0.2 | 0.2 | 0.5 | 0.3 | 0.2 | 0.2 | 0.5 | 0.3 |
| Executor | 14.8 | 10.2 | 34.6 | 26.9 | 15.4 | 10.5 | 36.0 | 27.5 |
| Total | 15.9 | 10.8 | 36.7 | 28.2 | 16.5 | 11.2 | 38.1 | 28.8 |

Fortran 90D compiler transforms programs and embeds CHAOS procedures in the translated codes. Performance of the compiler generated codes is compared with that of hand parallelized codes, in which appropriate CHAOS procedures are inserted by hand. All measurements were done on the Intel iPSC/860 machine. Initially, data arrays are in BLOCK distribution.

### 5.1 Effect of Irregular Distribution

A geometry based partitioner, recursive coordinate bisection (RCB) [1], was used to obtain an irregular data distribution. Performance results obtained using other kinds of partitioners are reported elsewhere [13].

The effect of irregular distribution is shown in Table 1. The table shows the performance of hand parallelized versions of the EUL3D template and the CHARMM template with irregular distribution and BLOCK distribution. *Partitioner* in the table represents the time needed to partition the arrays. *Executor* depicts the time needed to carry out the actual computation and communication for 100 iterations (time steps), and *Inspector* shows the time taken to build the communication schedule; *Remap* depicts the time taken to partition loop iterations and redistribute data. From the table, it can be seen that irregular distribution of arrays performs significantly better than the existing BLOCK distribution supported by HPF.

Table 3: Performance for Coordinate Bisection – EUL3D Loop

| Tasks (Time in Secs) | Hand | | | | Compiler | | | |
|---|---|---|---|---|---|---|---|---|
| | 10K Mesh | | 53k Mesh | | 10K Mesh | | 53k Mesh | |
| | Procs | | Procs | | Procs | | Procs | |
| | 8 | 16 | 32 | 64 | 8 | 16 | 32 | 64 |
| Partitioner | 0.3 | 0.4 | 2.4 | 2.0 | 0.3 | 0.4 | 2.5 | 2.0 |
| Remap | 1.1 | 0.6 | 2.6 | 1.6 | 1.2 | 0.8 | 2.6 | 1.7 |
| Inspector | 0.4 | 0.2 | 0.9 | 0.5 | 0.4 | 0.2 | 0.9 | 0.5 |
| Executor | 6.3 | 4.6 | 14.1 | 10.3 | 6.7 | 4.7 | 15.6 | 11.4 |
| Total | 8.1 | 5.8 | 20.0 | 14.4 | 8.6 | 6.1 | 21.6 | 15.6 |

### 5.2 Compiler Performance

Performance results that compare the costs incurred by the compiler-generated mapper coupler procedures with the costs of a hand embedded partitioner are presented.

Tables 2 and 3 present performance results of the Euler loop for both hand-coded and compiler-parallelized versions for various input mesh sizes. Table 2 presents performance for using BLOCK distribution, while Table 3 presents performance for using an irregular distribution obtained using the RCB partitioner. In the BLOCK version, each contiguous block of array elements is assigned to processors. Two important observations can be made from Tables 2 and 3. First, the compiler-generated code performs almost as well as the hand written code. In fact, the compiler generated code is within 15% of the hand coded version. The hand coded version performs better because the compiler generated code has to perform bookkeeping for the possibility of communication schedule reuse. Secondly, the use of a coordinate bisection partitioner leads to an improvement in the executor time by a factor of two compared to the use of block partitioning. The performance of the code with the irregular distribution is significantly better than the performance of the block partitioned code even when the cost of executing the partitioner is included.

### 5.3 Irregular Distribution via Reordering

This subsection presents performance results for the Euler solver template in Figure 14, in which the effect of irregular distributions are achieved by reordering array elements (Section 4.4). Recall that data arrays in the Euler solver code are accessed via integer indirection arrays. Initially, data arrays are BLOCK distributed. A coordinate partitioner is called and the result of partitioning is used to reorder data array elements. A procedure is then called to reorder indirection array values to match new data array element numbers. Note that this process does not involve redistributing data arrays.

Table 4 depicts performance results for the hand-parallelized version of the Euler solver template. The template is parallelized using CHAOS primitives and extrinsic HPF reordering library functions, *binary_dissection_2D*, *renumber_data_array* and *renumber_indirection_array*. All HPF extrinsic functions call CHAOS runtime support procedures to perform partitioning and reordering operations and a CHAOS primitive *scatter_add* is used to execute the intrinsic function *SUM_SCATTER*. The program shown in Figure 14 could be transformed by an HPF compiler by embedding calls to CHAOS primitives and extrinsic HPF reordering library functions. Since both the compiler-transformed code and the hand-parallelized version of the Euler solver template use the same set of CHAOS primitives and extrinsic HPF reordering library functions, the performance of the hand-parallelized code can provide a rough estimate of the performance that could be obtained by the code generated using an HPF compiler.

As shown in Figure 14, irregular loops are expressed as two-phase computations in HPF when indirectly accessed arrays appear on both left and right hand sides of statements. The two-phase computations result in two communication phases. Hence, two sets of communication schedules are generated. However, it seems plausible that the loop fusion [19] technique and sophisticated data flow analysis could be used by an HPF compiler to generate efficient code by combining the two computation phases as well as the two communication phases.

Table 4: Performance of Renumbering - Mesh Template - 53K Mesh

| (Time in Secs) | Naive | | Optimized | |
|---|---|---|---|---|
| | processors | | processors | |
| | 32 | 64 | 32 | 64 |
| Partitioner | 2.6 | 2.1 | 2.6 | 2.1 |
| Renumber | 0.7 | 0.5 | 0.7 | 0.5 |
| Inspector | 0.6 | 0.3 | 0.3 | 0.1 |
| Remap | 1.4 | 0.9 | 1.5 | 0.9 |
| Executor | 16.9 | 12.5 | 14.1 | 10.3 |
| Total | 22.2 | 16.3 | 19.2 | 14.0 |

In Table 4, *Partitioner* depicts the time required 1) to partition data arrays using a coordinate bisection partitioner and 2) to remap data based on the result of partitioning; *Renumber* depicts the time taken to renumber indirection arrays; *Remap* depicts time taken to partition loop iterations and redistribute indirection arrays; *Inspector* shows the time to compute communication schedules; *Executor* is the time taken to carry out the actual computation and communication. In the optimized version of the code, both computation and communication phases are executed in a single phase. In comparing the results for the optimized case presented in Table 4 with those of the hand-coded version for the coordinate bisection partitioner presented in Table 3, note that while the executor costs are the same, the pre-processing cost is slightly lower for the reordering technique. This difference is due to the face that the deference overhead of the optimized version is smaller since the deference operation is carried out with the new (BLOCK) data distributions.

## 6 Conclusions

This paper has presented methods that make it possible to efficiently support an important subclass of irregular problems using data parallel languages. The approach involved the use of a portable, compiler-independent, runtime support library called CHAOS. The CHAOS runtime support library contains procedures that

- support static and dynamic distributed array partitioning,
- partition loop iterations and indirection arrays,
- remap arrays from one distribution to another, and
- carry out index translation, buffer allocation and communication schedule generation.

The CHAOS runtime procedures are used by a Fortran 90D compiler to handle irregular distributions. Performance results of compiler-generated and hand-parallelized versions of an unstructured mesh computational fluid dynamics template and a molecular dynamics template were presented. The performance of the compiler-generated code is within 15% of that of the hand coded version.

A reordering method was described that makes it possible to support irregular distributions in HPF. Irregular distributions can be emulated in HPF by reordering elements of data arrays and renumbering indirection arrays. An irregular computational kernel was parallelized using CHAOS routines along with reordering and renumbering procedures. The results suggest that an HPF compiler could use reordering and renumbering extrinsic functions to obtain performance comparable to that achieved by a compiler for a language (such as Fortran 90D) that directly supports irregular distributions. This example kernel also served to illustrate that reordering is no panacea. In order to use the reordering method, users are forced to make numerous calls to extrinsic library functions.

## Acknowledgements

The authors thank Charles Koelbel for providing many insights into the applicability of HPF intrinsics and extrinsics for irregular problems; also Ken Kennedy, Seema Hiranandani and Sanjay Ranka for many useful discussions about integrating Fortran D runtime support for irregular problems.

The authors gratefully acknowledge Zeki Bozkus and Tom Haupt for the time they spent explaining the internals of the Fortran 90D compiler. The authors thank Robert Martino and DCRT for the general support and the use of NIH iPSC/860.

The authors thank Jim Humphries for his wonderful figures and Donna Meisel for proofreading this manuscript.

## References

[1] M.J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. on Computers*, C-36(5):570–580, May 1987.

[2] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. Compiling Fortran 90D/HPF for distributed memory MIMD computers. *Journal of Parallel and Distributed Computing*, 21(1):15–26, April 1994.

[3] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[4] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[5] 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.

[6] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

[7] Dennis Gannon, Shelby Yang, and Peter Beckman. *User Guide for a Portable Parallel C++ Programming System, pC++*. Department of Computer Science and CICA, Indiana University, January 1994.

[8] 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.

[9] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[10] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[11] D. J. Mavriplis. Three dimensional unstructured multigrid for the Euler equations, paper 91-1549cp. In *AIAA 10th Computational Fluid Dynamics Conference*, June 1991.

[12] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Proceedings Supercomputing '93*, pages 361–370. IEEE Computer Society Press, November 1993. Also available as University of Maryland Technical Report CS-TR-3055 and UMIACS-TR-93-32.

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

[14] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[15] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, December 1991.

[16] H. Simon. Partitioning of unstructured mesh problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Pergamon Press, 1991.

[17] A. Vidwans, Y. Kallinderis, and V. Venkatakrishnan. A new parallel dynamic load balancing algorithm for 3d adaptive unstructured grids. In *Proceedings of the 11th AIAA CFD Conference*, Orlando FL, July 1993.

[18] R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency, Practice and Experience*, 3(5):457–481, October 1991.

[19] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Machines.* Addision-Wesely, 1991.