1995

Supporting Irregular Distributions Using Data-Parallel Languages

Ravi Ponnusamy  
*University of Maryland*

Yuan-Shin Hwang  
*University of Maryland*

Raja Das  
*University of Maryland*

Alok Choudhary  
*Syracuse University*

Geoffrey Fox  
*Syracuse University*

Follow this and additional works at: [http://surface.syr.edu/npac](http://surface.syr.edu/npac)  
Part of the [Computer Sciences Commons](http://surface.syr.edu/npac/25)

**Recommended Citation**  
[http://surface.syr.edu/npac/25](http://surface.syr.edu/npac/25)

This Working Paper is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Northeast Parallel Architecture Center by an authorized administrator of SURFACE. For more information, please contact [surface@syr.edu](mailto:surface@syr.edu).
Supporting Irregular Distributions Using Data-Parallel Languages

Ravi Ponnusamy, Yuan-Shin Hwang, Raja Das, and Joel H. Saltz
University of Maryland at College Park
Alok Choudhary and Geoffrey Fox
Northeast Parallel Architectures Center, Syracuse University

Languages such as Fortran D provide irregular distribution schemes that can efficiently support irregular problems. Irregular distributions can also be emulated in HPF. Compilers can incorporate runtime procedures to automatically support these distributions.

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 such scheme is BLOCK distribution, which divides an array into contiguous, equal-sized subarrays and assigns each subarray to a different processor. Another is CYCLIC distribution, which assigns consecutively indexed array elements to processors in round-robin fashion.

However, more complex distributions are required to efficiently execute irregular problems such as computational fluid dynamics codes, molecular dynamics codes, diagonal or polynomial preconditioned iterative linear solvers, and time-dependent flame-modeling codes. Researchers have developed a variety of methods to obtain data mappings that optimize the communication requirements of irregular problems. These methods produce irregular distributions.

The Fortran D, Fortran 90D, and Vienna Fortran data-parallel languages support irregular data distributions. Fortran D and Fortran 90D let a programmer explicitly specify an irregular distribution using an array, to specify a mapping of array elements to processors. (Fortran D is Fortran 77 with data distribution; Fortran 90D is Fortran 90 with data distribution.) Vienna Fortran lets developers define functions to describe irregular distributions. However, the current version of High Performance Fortran does not directly support irregular distributions.

Also, in irregular problems, data-access patterns and workload are usu-
ally known only at runtime, so decisions regarding data and work distributions are made at runtime. These on-the-fly decisions therefore require special runtime support. Data-parallel languages do not provide this support, so we developed Chaos, a set of procedures that can be used by an HPF-style compiler to automatically manage programmer-defined distributions, partition loop iterations, remap data and index arrays, and generate optimized communication schedules. Other researchers have proposed compile-time techniques to partition data automatically, but their approaches only apply to regular programs.7

We implemented our methods on a Fortran 90D compiler, using templates from real application code for irregular problems. Our results show that using irregular distributions significantly improves performance, and that the compiler-generated code performs comparably to hand-parallelized versions of the same code. We also developed a method to emulate irregular distributions in HPF by reordering elements of data arrays and renumbering indirection arrays (which we'll discuss later). Our results suggest that an HPF compiler using this method will perform comparably to a compiler for a language (such as Fortran 90D) that directly supports irregular distributions.

**Irregular distributions**

Irregular problems extensively use indirectly accessed arrays. For example, Figure 1 illustrates code with an irregular loop. The code sweeps over edge 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.

**Figure 1. An irregular loop.**

**Figure 2. Data distributions:** (a) example graph, (b) BLOCK distribution, (c) CYCLIC distribution, (d) irregular distribution. Dashed circles indicate that the indexed elements are not local.
The edges are represented by two indirection arrays `edge1` and `edge2`, which will be partitioned in blocks. The code in Figure 1 can be used to sweep this graph.

The BLOCK distribution (Figure 2b) assigns nodes 1, 2, and 3 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 nonlocal. The BLOCK distribution has four nonlocal data elements; the CYCLIC distribution (Figure 2c) has five. The irregular distribution (Figure 2d) represents the best mapping: It requires only one remote reference.

**Language support**

Vienna Fortran, pC++, Fortran D, Fortran 90D, and HPF provide a rich set of directives that let programmers specify desired data decompositions. With these directives, compilers can partition loop iterations and generate the communication required to parallelize the code. Although we focus on Fortran D, Fortran 90D, and HPF, this research could be extended to other languages. (In the program code examples in this article, Fortran D, Fortran 90D, and HPF directives are in all capital letters; programmer-declared variables are lower-case.)

**Fortran D and Fortran 90D**

In Fortran D and Fortran 90D, the `DECOMPOSITION`, `ALIGN`, and `DISTRIBUTE` directives provide explicit control over data partitioning. A template, called a `distribution`, is declared and used to characterize a distributed array's significant attributes. A distribution is produced using two declarations. The first is `DECOMPOSITION`, which fixes the template's name, dimensions, and size. The second is `DISTRIBUTE`, which is an executable statement that specifies how the template will be mapped onto the processors. Programmers can choose from several regular distributions, and can explicitly specify how a distribution is to be mapped onto the processors. The `ALIGN` statement associates a specific array with a distribution.

In the Fortran D code segment in Figure 3, `d` is declared to be a 2D decomposition of size $N \times N$. Array `a` is then aligned with the decomposition `d`. Distributing decomposition `d` by `(*, BLOCK)` produces a column partition of arrays aligned with `d`. The data-distribution specifications are then treated as comment statements in a sequential-machine Fortran compiler. So, a program written with distribution specifications can be compiled and executed on a sequential machine.

**Support for irregular distributions**

Fortran D and Fortran 90D support irregular data distributions and dynamic data decomposition—that is, changing a decomposition's alignment or distribution at any point in the program. An irregular partition of distributed array elements can be explicitly specified. Figure 4 depicts such a declaration in Fortran D. Statement S3 defines two 1D decompositions, each of size N. Statement S4 partitions decomposition `reg` into equal-sized blocks, with one block assigned to each processor. Statement S5 aligns array `map` with distribution `reg`. In statement S7, `map` specifies how distribution `irreg` will be partitioned. 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.

**Computational loop structures**

Figure 5 shows an irregular Fortran 90D `FORALL` loop that is equivalent to the sequential loop `L2` in Figure 1. `L2` represents a sweep over the edges of an unstructured mesh. Because the mesh is unstructured, an indirection array must be used to access the vertices during a loop over the edges. In `L2`, the reference pattern is specified by integer arrays `edge1` and `edge2`. `L2` carries out reduction operations that are the only types of dependence between different iterations of the loop. For example, in `L2` each mesh vertex is updated using the
corresponding values of its neighbors (directly connected through edges). Each vertex is updated as many times as the number of neighboring vertices. Fortran D and Fortran 90D’s implementation of FORALL follows copy-in-copy-out semantics; loop-carried dependences are not defined. The present implementation allows loop-carried dependences caused by reduction operations. The reduction operations are specified in a FORALL construct using the REDUCE construct. Reduction inside a FORALL is important for representing computations such as those in sparse and unstructured problems. This representation also preserves the explicit parallelism available in the underlying computations.

**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 called the owner-computes rule. If the element on the left 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 dependences. For example, consider the following Fortran D loop:

```fortran
C 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 dependences. If work is assigned using the owner computes rule, for iteration \( i \), statement \( S1 \) would be computed on the owner of \( x(ib(i)) \), \( \text{OWNER}(x(ib(i))) \), while statement \( S2 \) would be computed on the owner of \( y(ia(i)) \), \( \text{OWNER}(y(ia(i))) \). The value of \( x(ib(i)) \) would have to be communicated whenever \( \text{OWNER}(x(ib(i))) \neq \text{OWNER}(y(ia(i))) \).

In Fortran D and Fortran 90D a programmer can use the ON clause to specify which processor will carry out a loop iteration. For example, in Fortran D, a loop could be written as

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

Iteration \( i \) must be computed on the processor on which \( x(i) \) resides, if the sizes of arrays \( ia \) and \( ib \) are equal to the number of iterations. A similar proposed HPF directive, \text{EXECUTE-ON-HOME}, provides this capability.

Another method uses the almost-owner-computes rule, which executes a loop iteration on the processor that is the home of the largest number of distributed array references in that iteration. For example, in the Fortran D code segment

```fortran
C$ EXECUTE (i) ON-HOME(map(i))
FORALL i = 1, N
S1 x(ib(i)) =........
S2 y(ia(i)) = x(ib(i))
END FORALL
```

This loop uses the proposed HPF \text{EXECUTE-ON-HOME} directive to present the almost owner computes rule. An iteration \( i \) is assigned to the processor \( \text{map}(i) \).
Two irregular problems

The loop structures of two application codes—an unstructured Euler solver and a molecular dynamics code—illustrate the need for irregular distributions. These structures consist of a sequence of loops with indirectly accessed arrays, and are similar to the loop in Figure 1 in the main article.

Unstructured Euler solver

The first application code is an unstructured Euler solver used to study the flow of air over an airfoil.1 Complex aerodynamic shapes require high-resolution meshes and, consequently, large numbers of mesh points. Physical values (such as velocity and pressure) are associated with each mesh vertex. These values are called flow variables and are stored in 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 (that is, arrays that store flow variables) must be partitioned. Because meshes are typically associated with physical objects, a spatial location can often be associated with each mesh point. The mesh-generation strategy determines the spatial locations of the mesh points and the connectivity pattern (edges) of the vertices. Figure A shows an unstructured mesh of a 3D aircraft wing that was generated by such a process.

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 criteria. This frequently produces a vertex numbering that does not correspond usefully with the edge numbering. One way to solve this is to renumber the mesh completely after it has been generated.

Mesh points are partitioned to minimize communication. Some promising, recent partitioning heuristics use one or several of these types of information: the spatial locations of mesh vertices, the connectivity of the vertices, and an estimate of the computational load associated with each mesh point. For instance, a developer might choose a partitioner that is based on coordinates. A coordinate bisection partitioner decomposes data using the spatial locations of mesh vertices. If the developer 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 to balance the load. Therefore, mesh edges are partitioned so that load balance is maintained and computations employ mostly locally stored data.

Molecular dynamics code

Other unstructured problems have similar indirectly accessed arrays. For instance, consider the nonbonded force calculation in the molecular dynamics code, Charmm (see Figure B). Force components associ-

A programmer-defined function determines the values of array map.

Figure 6 depicts an irregular loop from the Charmm molecular dynamics code (see the sidebar) in Fortran 90D with the EXECUTE-ON-HOME directive for partitioning loop iterations. The inner loop iterations are executed on processors that own \texttt{reg}(i), where \texttt{reg} is the decomposition to which arrays \texttt{x}, \texttt{y}, \texttt{dx}, and \texttt{dy} are aligned. The array \texttt{inblo} is replicated on all processors.

\begin{verbatim}
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
\end{verbatim}

Figure A. An unstructured mesh of a 3D aircraft wing.

Figure B. Nonbonded force calculation loop from Charmm.

High Performance Fortran

Although the current version of HPF does not support nonstandard distributions, it can indirectly support such distributions by reordering array elements to reduce communication requirements. Applications scientists have frequently employed variants of this approach when porting irregular codes to parallel architectures. First, a partitioner maps array elements to processors. Next, array elements are reordered so that elements mapped to a given processor are assigned to consecu-
ated with each atom are stored as Fortran arrays. The loop L1 sweeps over all atoms. We'll assume that L1 is a parallel loop and L2 is sequential. The loop iterations of L1 are distributed over processors. All computation for iteration i of L1 is performed on a single processor, so loop L2 need not be parallelized.

We assume that all atoms within a given cutoff radius interact. The array Partner(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 atom j.

The force array elements are partitioned to reduce interprocessor communication in the nonbonded force calculation loop (Figure B). Figure C depicts two possible distributions of the atoms of a Myoglobin molecule and 3830 water molecules onto eight processors. Shading represents the assignment of atoms to processors. Data sets associated with the sequential version of Charmm assign each atom an index number that does not reflect locality. Figure C1 depicts a distribution that assigns consecutively numbered sets of atoms to each processor—that is, a BLOCK distribution. Because nearby atoms interact, a BLOCK distribution will likely cause a large volume of communication. Figure C2 depicts a distribution based on the spatial locations of atoms. An inertial bisection partitioner performs the distribution, which produces much less surface area between the portions of the molecules associated with each processor.

References

Figure C. Distribution of atoms on eight processors: (1) BLOCK distribution, (2) irregular distribution.

The graph shows the flow of data between elements of arrays x and y; an edge between nodes n1 and n2 means the value of x(n1) is accumulated to y(n2) and the value of x(n2) is accumulated to y(n1).

Partitioning should occur to allocate the same number of nodes to processors, and to minimize the number of cross-edges between processors—that is, to minimize the number of edges where both end-nodes are not on the same processor. In Figure 7b 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 (3,1).

Figure 7c shows a better distribution, with a smaller number of cross-edges. A partitioner assigns nodes 1, 2, and 5 to processor 0, and the rest to processor 1; there is only one cross-edge, (2,3). The partitioner produces an arbitrary assignment of nodes to processors—that is, an irregular distribution.

The effects of this distribution can be obtained by assigning new indices to the nodes so that contiguously numbered nodes are assigned to each processor. This renumbering transforms the graph in Figure 7c (an irregular distribution) to that in Figure 7d (a BLOCK distribution). Figure 7c and Figure 7d are partitioned identically; but their nodes (and consequently, their edges) are numbered differently. So, the cross-edge in Figure 7d is (2,4).

Extrinsic procedures are called to invoke the partitioners and to reorder the data and renumber the indirection arrays. Using the EXTRINSIC directive, a non-HPF procedure can be interfaced with HPF programs (see Figure 8). For example, Statement S2 specifies the interface from HPF to a partitioner binary_dissection_2D. The directive HPF_LOCAL indicates that the procedure binary_dissection_2D has been written in local HPF style. This procedure uses information provided in arrays x and y and writes the result of the partitioning to the permutation array reorder. Statements S3 and S4 specify the input (x and y) and output (reorder) parameters.

Figure 9 illustrates the reordering technique in HPF for a Euler solver template (see the sidebar). First, arrays x, y, and reorder are distributed by BLOCK. Next, an extrinsic partitioner procedure is called to determine the values of reorder. An extrinsic procedure, renumber_data_array, is invoked to reorder x and y based on the values of reorder. After the reordering is completed, the rth element of x is moved to the position reorder(i), and another extrinsic function, renumber_indirection_array, is called to update arrays edge1 and edge2 so that their values reflect the new positions of the array elements of x and y; that is, the value of edge1(i) is modified to reorder(edge1(i)).

The current version of HPF does not support Fortran D's REDUCE construct. 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 9 depicts a method of expressing the irregular loop L2 in HPF. Here, the HPF intrinsic function SUM_SCATTER expresses an array-combining operation. A statement in a sequential irregular loop that has indirectly accessed arrays on the statement's right and left, can be written as two separate phases: a FORALL loop to carry out the computation on the right and store the values to a temporary array temp, and an intrinsic function SUM_SCATTER to scatter and combine the elements of temp to array y.

Padding and Reordering

The preceding discussion assumes that the number of array elements can be evenly divided by the number of processors, and that 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 to balance the workload. To accomplish this, the programmer first declares the original array as an oversized array (in BLOCK distribution); this is called padding the array. Next, a partitioner is called to assign the array elements to processors so that no more than a given number of elements are assigned to any processor.

Let's assume that a 1D 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 programmer decides that no more than $M$ ($M > N$) array elements can be assigned to any processor.

1. The programmer declares A as an $M \times P$ BLOCK-distributed array. Originally, only the first $N \times P$ elements of A are initialized with meaningful values, and the last $(M-N) \times P$ elements of A are unused storage.
2. The programmer 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 $A(reorder(i))$, where $1 \leq i \leq N \times P$. To assign $A(i)$ to processor $p$, where $0 \leq p < P$, the partitioner defines reorder(i) as $M \times (p - 1) < reorder(i) \leq M \times P$. 18
INTERFACE
EXTRINSIC(HPF_LOCAL) SUBROUTINE binary_dissection_2D(reorder, x, y, n)
REAL*8, DIMENSION(:), INTENT(IN) :: x, y
INTEGER INTENT(IN) :: n
INTEGER, DIMENSION(:), INTENT(OUT) :: reorder
END SUBROUTINE binary_dissection_2D
END INTERFACE

Figure 8. Interfacing an extrinsic partitioner procedure.

!HPF$ TEMPLATE reg(N), regl(M)
!HPF$ DISTRIBUTE(BLOCK) ONTO P :: reg, regl
!HPF$ ALIGN WITH reg :: x, y, reorder
!HPF$ ALIGN WITH regl :: edgel, edge2, temp

use an extrinsic partitioner procedure to obtain reorder array
CALL binary_dissection_2D(reorder, x, y, n-local)
use an extrinsic procedure to reorder data arrays
CALL renumber_data_array(reorder, x, n-local)
CALL renumber_data_array(reorder, y, n-local)
use an extrinsic procedure to renumber indirection arrays
CALL renumber_indirection_array(reorder, edgel, n-local)
CALL renumber_indirection_array(reorder, edge2, n-local)

Figure 9. Irregular distribution and loops in HPF.

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

For example, an array with 8 meaningful elements (see Figure 10a) is declared as a 10-element BLOCK array (see Figure 10b). Figure 10c depicts the result of a reordering based on the reorder array returned by a partitioner. The 4th element of A is moved to position reorder(4); for example, when reorder(1) = 6, A(1) in Figure 10b is moved to A(6) in Figure 10c. There are two ghost elements (in dashed lines) at the middle of the reordered array.

Runtime support

We developed the Chaos runtime support library, a superset of the Parti library, to efficiently handle problems that consist of a sequence of clearly demarcated concurrent computational phases. With Chaos, solving such irregular problems on distributed-memory machines involves six major phases:

1. Data partitioning assigns elements of data arrays to processors.
2. Data remapping redistributes data-array elements.
3. Iteration partitioning allocates iterations to processors.
4. Iteration remapping redistributes indirection array elements.
5. Inspector translates indices and generates communication schedules.
6. Executor uses schedules for data transportation, and performs computation.

Spring 1995
Initially, arrays are decomposed into either regular or irregular distributions. The first four phases map data and computations onto processors. The next two analyze data-access patterns in a loop and generate optimized communication calls.

The sixth phase (executor) typically occurs many times in real application codes; however, the first four phases are executed only once if the data-access patterns do not change. When programs change data-access patterns but maintain good load balance, inspector and executor are repeated. If programs require remapping of data arrays from the current distribution to a new distribution, all phases are executed again. We'll now look at these phases in more detail.

**Data partitioning**

Data partitioning uses partitioners provided by Chaos or the programmer. Chaos supports a number of parallel partitioners that use heuristics based on spatial positions, computational load, connectivity, and so on. 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 that lists the home processor and offset address of each data array element.

The translation table has the following fields:
- Global size \( N \)
- Distribution type \( T \)
- Block size \( B \)
- Local size \( L \)
- Processor list \( p \)
- Offset list \( I \)

The first four fields represent regular distributions such as **BLOCK** and **CYCLIC**. The processor list and offset list fields represent irregular distributions. The processor list gives the home processor of each array element; the offset list gives the local addresses of the elements.

To access an element \( A(m) \) of distributed array \( A \), a translation table lookup is necessary to determine the location of \( A(m) \). This lookup, which is aimed at computing the home processor and the offset associated with a global distributed array index, is called a dereference request. Any preprocessing to optimize communication must perform dereferencing, because 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. Table-lookup costs are the primary consideration in adaptive problems, because 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 is called a **replicated translation table**. The storage cost for this 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, because each processor has an identical translation table.

Because of memory considerations, it is not always feasible to place a copy of the translation table on each processor. In this case, the translation table can be distributed between processors. This is called a **distributed translation table**. Earlier versions of Parti supported a translation table that was distributed by blocks; the first \( N/P \) elements were put on the first processor, the second \( N/P \) elements were put on the second processor, and so on.

In Chaos, 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 \( I((m-1)/N) \times P + 1 \). Distributed translation tables have the highest use of available distributed memory for a fixed-size irregularly distributed array. The dereference requests, however, might require a communication step because some portions of the translation table do not reside in the local memory. Similarly, table reorganization also requires interprocessor communication because each processor is authorized to modify only a limited portion of the translation table.

Chaos also supports an intermediate degree of replication with **paged translation tables**. This scheme divides the translation table into pages, which are distributed across processors. Processors that frequently refer to a page receive a copy of the page, making subsequent references local.

Figure 11 depicts the three translation-table structures of a graph partitioned over two processors. Only the processor list \( p \) and offset list \( l \) are displayed. The numbers above arrays are the index numbers of nodes. Figure 11a shows an irregular distribution. Nodes 1, 2, and 5 are assigned to processor \( P_0 \), and nodes 3, 4, and 6 to processor \( P_1 \). The distributed translation table (Figure 11b) assigns the first three elements of \( p \) and \( l \) on \( P_0 \) and the last three on \( P_1 \). By contrast, the replicated translation table (Figure 11c) replicates all the six elements of \( p \) and \( l \) on both processors. The paged translation table (Figure 11d) has a page size of two; each processor owns two pages. The dashed page on \( P_0 \) is
copied from $P_1$ as the result of remote references of node 5 from $P_0$ to $P_1$.

**Data Remapping**

For efficiency, distribution of data arrays may have to change between computational domains or phases. For instance, as computation progresses in an adaptive problem, the workload and distributed-array access patterns may change based on the nature of problem. This might cause poor load balance among processors. So, 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, $\delta_k$. Then, a heuristic method produces an irregular distribution $\delta_h$. Once the new distribution is obtained, all data arrays associated with distribution $\delta_k$ must be transformed to distribution $\delta_h$.

To redistribute data, a runtime procedure called remap takes as input the original and the new distribution in the form of translation tables, and returns a communication schedule (which we'll discuss later) that is used to move data between initial and subsequent distributions.

**Iteration Remapping**

Iteration remapping is similar to data remapping. Indirection array elements are remapped (by the remap procedure) to conform with the loop iteration partitioning. For example, in Figure 1, once loop $L_2$ is partitioned, the indirection array elements $\text{edge1}(i)$ and $\text{edge2}(i)$ used in iteration $i$ are moved to the processor that executes that iteration.

**Inspector**

The inspector carries out the preprocessing needed for communication optimizations and index translation. This phase also builds a communication schedule, which is used for data transportation and computation. Communication schedules determine the number of communication startups and the volume of communication, so it is important to optimize them.

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

- **Send list**: a list of arrays that specifies the local elements of a processor $p$ required by all processors.
- **Permutation list**: an array that specifies the data-placement order of off-processor elements in the local buffer of processor $p$.
- **Send size**: an array that specifies sizes of outgoing messages from processor $p$ to all processors.
- **Fetch size**: an array that specifies sizes of incoming messages to processor $p$ from all processors.
Table 1. Irregular distribution’s effect on performance (in seconds) of hand-parallelized code, 32 processors. Eu13D (a Euler solver) is a 53K mesh; the Charmm data set is 14K atoms.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>COORDINATE BISECTION</th>
<th>BLOCK PARTITION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Eu13D</td>
<td>CHARMM</td>
</tr>
<tr>
<td>Partitioning</td>
<td>2.4</td>
<td>0.7</td>
</tr>
<tr>
<td>Remapping</td>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td>Inspector</td>
<td>0.9</td>
<td>0.7</td>
</tr>
<tr>
<td>Executor</td>
<td>14.1</td>
<td>93.5</td>
</tr>
<tr>
<td>Total</td>
<td>20.0</td>
<td>97.4</td>
</tr>
</tbody>
</table>

Table 2. Performance (in seconds) for BLOCK distribution of the Euler solver.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>10K MESH</th>
<th>53K MESH</th>
<th>10K MESH</th>
<th>53K MESH</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HAND</td>
<td>COMPILER</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Processors</td>
<td>Processors</td>
<td>Processors</td>
<td>Processors</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Partitioning</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Remapping</td>
<td>0.9</td>
<td>0.4</td>
<td>1.6</td>
<td>1.0</td>
</tr>
<tr>
<td>Inspector</td>
<td>0.2</td>
<td>0.2</td>
<td>0.3</td>
<td>0.2</td>
</tr>
<tr>
<td>Executor</td>
<td>14.0</td>
<td>10.2</td>
<td>34.6</td>
<td>26.9</td>
</tr>
<tr>
<td>Total</td>
<td>15.9</td>
<td>10.8</td>
<td>36.7</td>
<td>28.2</td>
</tr>
</tbody>
</table>

**EXECUTOR**

The executor uses information from the earlier phases to carry out computation and communication. The Chaos *gather* and *scatter* data-transportation primitives use the communication schedules to move data. *Gather* fetches a copy of off-processor elements into a local buffer; *scatter* sends off-processor elements back to their home processors after computation.

**Compiler support and experimental results**

We incorporated runtime support for irregular distributions in the Fortran 90D compiler being developed at Syracuse University. The compiler transforms programs and embeds Chaos procedures in the translated codes. We tested compiler transformations for irregular templates in Charmm and Eu13D, a loop from a Euler solver (see the sidebar). Here we'll compare performance of the compiler-generated code to hand-parallelized code, where appropriate Chaos procedures are inserted by hand. We'll also see how effective the HPF reordering technique is. All measurements were made on an Intel iPSC/860 machine. Initially, data arrays were in BLOCK distribution.

**Effect of irregular distribution**

We used recursive coordinate bisection, a geometry-based partitioner, to obtain an irregular data distribution. Performance results for other kinds of partitioners are reported elsewhere.

**COMPILER PERFORMANCE**

Tables 2 and 3 present the performance of hand-coded and compiler-parallelized versions of the Euler loop for two input mesh sizes. Table 2 is for a BLOCK distribution; Table 3 is for an irregular distribution using the recursive coordinate bisection partitioner. We draw two important conclusions from the results. First, the compiler-generated code performs almost as well (within 15%) as the hand-written code. The hand-coded version performs better because the compiler-generated code has to perform bookkeeping for possible communication schedule reuse. Second, the coordinate bisection partitioner improves executor time by a factor of two compared to BLOCK partitioning. The code with the irregular distribution performs significantly better than the BLOCK-partitioned code, even when the cost of executing the partitioner is included.

**Irregular distribution by reordering**

We'll now examine the performance of the Euler solver template in Figure 9, which achieved the effect of irregular distributions by using a partitioner, by reordering array elements, and by renumbering indirection arrays. This process did not involve redistributing data arrays.

We hand-parallelized the Euler solver template 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 reorder-
Table 3. Performance (in seconds) for coordinate bisection partitioning of the Euler solver.

<table>
<thead>
<tr>
<th>TASKS</th>
<th>HAND</th>
<th>COMPILER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10K mesh</td>
<td>53K mesh</td>
</tr>
<tr>
<td></td>
<td>8 processors</td>
<td>16 processors</td>
</tr>
<tr>
<td>Partitioning</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>Mapping</td>
<td>1.1</td>
<td>0.6</td>
</tr>
<tr>
<td>Inspector</td>
<td>0.4</td>
<td>0.2</td>
</tr>
<tr>
<td>Executor</td>
<td>6.3</td>
<td>4.6</td>
</tr>
<tr>
<td>Total</td>
<td>8.1</td>
<td>5.8</td>
</tr>
</tbody>
</table>

Table 4. Performance (in seconds) of a renumbered Euler solver template, 53K mesh.
The native version has two computation and two communication phases, and the optimized version has one computation and one communication phase.

<table>
<thead>
<tr>
<th>TASKS</th>
<th>NATIVE</th>
<th>OPTIMIZED</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32 processors</td>
<td>64 processors</td>
</tr>
<tr>
<td>Partitioning</td>
<td>2.6</td>
<td>2.1</td>
</tr>
<tr>
<td>Renumbering</td>
<td>0.7</td>
<td>0.5</td>
</tr>
<tr>
<td>Remapping</td>
<td>1.4</td>
<td>0.9</td>
</tr>
<tr>
<td>Inspector</td>
<td>0.6</td>
<td>0.3</td>
</tr>
<tr>
<td>Executor</td>
<td>16.9</td>
<td>12.5</td>
</tr>
<tr>
<td>Total</td>
<td>22.2</td>
<td>16.3</td>
</tr>
</tbody>
</table>

Althouh appropriate irregular data distributions reduce the communication requirements of irregular scientific programs, it is tedious for programmers to write explicit parallel programs that handle irregular distributions and manage interprocessor messages. The Chaos runtime library is designed to relieve users of such a burden. It supports a programming environment with a global name space and provides efficient functions for collective communication operations and index translations that conform to the current data distributions. The functions can be manually embedded in explicit parallel programs by users, or automatically inserted into single-program-multiple-data (SPMD) programs that are transformed by data-parallel compilers.

Data-parallel languages provide programmers a single-threaded and loosely synchronous programming environment with a global name space. Users can easily specify the appropriate data distributions for applications if data-parallel languages support irregular distributions. Both Fortran D and Vienna Fortran provide directives to specify irregular distributions, but the current version of HPF only supports regular distributions.

However, the HPF Forum has set up a group to exploit the possibility of incorporating irregular distributions.
butions in the next version of HPF (HPF2). Before HPF2 compilers are available, users can apply the reordering and array-padding techniques described in this article to simulate irregular distributions in HPF.

FURTHER INFORMATION

ACKNOWLEDGMENTS
This work was sponsored in part by ARPA (NAC 1-1685), NSF (ASC-9211921), and ONR (SC292-1-22913). We thank Charles Koelbel for providing many insights into the applicability of HPF intrinsics and extrinsics for irregular problems we also thank Ken Kennedy, Semaan Hiranandani, and Sanjay Ranka for many useful discussions about integrating Fortran D runtime support for irregular problems. We gratefully acknowledge Zeki Bockus and Tom Haupt for the time they spent explaining the internals of the Fortran D compiler. We thank Robert Martino and DCRT for the general support and the use of NIH iPSC/860. We also thank Jim Humphries for his wonderful figures and Donna Mouril for proofreading the manuscript.

REFERENCES
10. Alok Choudhary was an associate professor in the Department of Electrical and Computer Engineering at Syracuse University. His research interests are parallel and distributed processing, software development environments for parallel computers, including compilers and runtime support; parallel architectures, and parallel I/O systems. He has published over 60 journal and conference papers, and has authored Parallel Architectures and Parallel Algorithms for Integrated Vision Systems, published by Kluwer Academic Publishers. He is currently a research professor for the Journal of Parallel and Distributed Computing, and has been a guest editor for that journal and for Computer. He was program cochair for the 1993 International Conference on Parallel Processing. He received an IEEE Engineering Foundation Award in 1990, and the NSF Young Investigator Award in 1993.
11. Ravi Ponnusamy's research interests include parallel I/O, parallelizing compilers, supercomputer applications, and performance evaluation. He has been designing and developing tools and techniques for High Performance Fortran compilers to produce efficient parallel code for large-scale scientific applications. He coauthored a paper that received the Best Student Paper award at Supercomputing '92. He received his PhD in computer science from Syracuse University in 1994, and his BE in computer science and engineering from Anna University, Madras, India, in 1987.
12. Yuan-Shin Hwang is pursuing his PhD in computer science at the University of Maryland, College Park, where he is a research assistant in the High-Performance Systems Software Laboratory. His research interests include parallel and distributed computing, parallel architectures and compilers, and runtime support for sparse and unstructured scientific computations targeted to massively parallel supercomputers. He received his MS and BS in electrical engineering from the National Tsing Hua University, Hsinchu, Taiwan, in 1989 and 1987. He can be reached at the Dept. of Computer Science, Univ. of Maryland, College Park, M/D 20/42; Internet: shyn@cs.umd.edu.
13. Raja Das is a postdoctoral research associate with the Department of Computer Science and the Institute for Advanced Computer Studies at the University of Maryland, College Park. He is working on the development of compilers and runtime support for parallelizing sparse, unstructured, and adaptive scientific problems. He received his PhD in computer science from the College of William and Mary in 1993, his MS in mechanical engineering from Clemson University in 1987, and his BS in mechanical engineering from Jadavpur University in 1984. He can be reached at UMIACS and the Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742; Internet: raja@cs.umd.edu.
14. Joel II. Saltz is an associate professor with the Department of Computer Science and the Institute for Advanced Computer Studies, and is the director of the High Performance Systems Software Laboratory, at the University of Maryland, College Park. He leads a research group that is developing methods for producing portable compilers that generate efficient multiprocessor code for irregular scientific problems—that is, problems that are unstructured, sparse, adaptive, or block structured. Previously, he was lead computer scientist at the Institute for Computer Applications in Science and Engineering at the NASA Langley Research Center. He can be reached at UMIACS and the Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742; Internet: saltz@cs.umd.edu.
15. Geoffrey Fox is a professor of computer science and physics at Syracuse University and the director of the Northeast Parallel Architectures Center. He leads a project to develop prototypes of high performance Fortran compilers. He is also a leading proponent for the development of computational science as an academic discipline and a scientific method. His research has focused on the development and use of parallel computing to solve large-scale computational problems, and currently focuses on large-scale distributed multimedia information systems. Fox directs InfoMall, which focuses on accelerating the introduction of parallel computing into New York State industry. He coauthored Solving Problems on Concurrent Computers, and edits Concurrency: Practice and Experience and the International Journal of Modern Physics: C (physics and computers). He has served as an educational computing consultant and as assistant provost for computing at Caltech. He earned his PhD in theoretical physics from Cambridge University in 1967. He can be reached at the Northeast Parallel Architectures Center, Syracuse Univ., Syracuse, NY 13244; Internet: geoff@npac.syr.edu.