Runtime Support and Compilation Methods for User-Specified Irregular Data Distributions

Ravi Ponnusamy, Joel Saltz, Alok Choudhary, *Member, IEEE Computer Society*, Yuan-Shin Hwang, and Geoffrey Fox

***Abstract*—This paper describes two new ideas by which a High Performance Fortran compiler can deal with irregular computations effectively. The first mechanism invokes a user specified mapping procedure via a set of proposed compiler directives. The directives allow use of program arrays to describe graph connectivity, spatial location of array elements, and computational load. The second mechanism is a conservative method for compiling irregular loops in which dependence arises only due to reduction operations. This mechanism in many cases enables a compiler to recognize that it is possible to reuse previously computed information from inspectors (e.g., communication schedules, loop iteration partitions, and information that associates off-processor data copies with on-processor buffer locations). This paper also presents performance results for these mechanisms from a Fortran 90D compiler implementation.**

***Index Terms*—Runtime support, parallelizing compilers, data distributions, high performance Fortran, language directives, irregular problems, distributed memory machines.**

I. Introduction

A. Background

THIS paper addresses a class of irregular problems that consists of a sequence of clearly demarcated concurrent computational phases where patterns of data access and computational cost cannot be anticipated until runtime. In this class of problems, once runtime information is available, data access patterns are known before each computational phase. These problems are called *irregular concurrent problems* [9]. Examples of irregular concurrent problems include adaptive and self-adaptive explicit, multigrid unstructured computational fluid dynamic solvers [29], [15], molecular dynamics codes (CHARMM [5], AMBER [43], GROMOS [40], etc.), diagonal or polynomial preconditioned iterative linear solvers [41], and time dependent flame modeling codes [32].

This paper focuses on the runtime support, the language extensions, and the compiler support required to provide efficient data and work load distributions. The paper also presents methods and a prototype implementation that make it possible for compilers to efficiently handle irregular problems coded using a set of language extensions closely related to Fortran D [14], Vienna Fortran [45], and High Performance Fortran (HPF) [19].

The optimizations that must be carried out to solve irregular concurrent problems efficiently on a distributed memory machine include:

1) data partitioning,
2) partitioning computational work,
3) software caching methods to reduce communication volume, and
4) communication vectorization to reduce communication startup costs.

Since data access patterns are not known in advance, decisions about data structure and workload partitioning have to be deferred until runtime. Once data and work have been partitioned between processors, prior knowledge of loop data access patterns makes it possible to predict which data need to be communicated between processors. This ability to predict communication requirements makes it possible to carry out communication optimizations. In many cases, communication volume can be reduced by prefetching only a single copy of each referenced off-processor datum. The number of messages can also be reduced by using data access pattern knowledge to allow prefetching quantities of off-processor data. These two optimizations are called *software caching* and *communication vectorization*.

Whenever there is a possibility that a loop's data access patterns might have changed between consecutive loop invocations, it is necessary to repeat the preprocessing needed to minimize communication volume and startup costs. When data access patterns change, it may also be necessary to repartition computational work. Fortunately, in many irregular concurrent problems, data access patterns change relatively infrequently. This paper presents simple conservative techniques that in many cases make it possible for a compiler to verify that data access patterns remain unchanged between loop invocations, making it possible to amortize the associated costs of software caching and communication vectorization.

Fig. 1 illustrates a simple sequential Fortran irregular loop (loop L2) which is similar in form to loops found in unstructured computational fluid dynamics (CFD) codes and molecular dynamics codes. In Fig. 1, arrays `x` and `y` are accessed by indirection arrays `edge1` and `edge2`. Note that the data access pattern associated with the inner loop L2 is determined by integer arrays `edge1` and `edge2`. Because arrays `edge1` and `edge2` are not modified within loop L2, L2's data access pattern can be anticipated prior to executing

Manuscript received Dec. 31, 1993; revised Sept. 26, 1994.

R. Ponnusamy is with the UMIACS and Computer Science Department, University of Maryland, College Park, MD 20742 and the Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY 13244.

J. Saltz and Y.-S. Hwang are with the UMIACS and Computer Science Department, University of Maryland, College Park, MD 20742.

A. Choudhary and G. Fox are with the Northeast Parallel Architectures Center, Syracuse University, Syracuse, NY 13244.

IEEECS Log Number D95010.

L2. Consequently, `edge1` and `edge2` are used to carry out preprocessing needed to minimize communication volume and startups. Whenever it can be determined that `edge1`, `edge2`, and `nedges` have not been modified between consecutive iterations of outer loop L1, repeated preprocessing can be avoided.

```
C Outer Loop L1
  do n = 1, n_steps
  ...
C Inner Loop L2
   do i = 1, nedges
     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
```

Fig. 1. An example code with an irregular loop.

B. Irregular Data Distribution

On distributed memory machines, large data arrays need to be partitioned between local processor memories. These partitioned data arrays are called *distributed arrays.* Long term storage of distributed array data is assigned to specific processor and memory locations in the machine. Many applications can be efficiently implemented by using simple schemes for mapping distributed arrays. One example of such a scheme would be the division of an array into equal sized contiguous subarrays and assignment of each subarray to a different processor. Another example would be to assign consecutively indexed array elements to processors in a round-robin fashion. These two data distribution schemes are often called BLOCK and CYCLIC data distributions [13], respectively.

Researchers have developed a variety of heuristic methods to obtain data mappings that are designed to optimize irregular problem communication requirements [39], [44], [27], [25], [3], [17]. The distribution produced by these methods typically results in a table that lists a processor assignment for each array element. This kind of distribution is often called an *irregular distribution.*

Partitioners typically make use of one or more of the following types of information:

1) a description of graph connectivity,
2) spatial locations of array elements, and
3) information that associates array elements with computational load.

Languages such as HPF, Fortran D, and Vienna Fortran allow users to advise the compiler of how array elements should be assigned to processor memories. In HPF a pattern of data mapping can be specified using the `DISTRIBUTE` directive. Two major types of patterns can be specified this way: `BLOCK` and `CYCLIC` distributions. For example,

```
REAL, DIMENSION(500,500) :: X, Y
!HPF $ DISTRIBUTE (*, BLOCK) :: X
!HPF $ DISTRIBUTE (BLOCK, BLOCK) :: Y
```

breaks the arrays X and Y into groups of columns and rectangular blocks, respectively.

This paper describes an approach where the user does not *explicitly* specify a data distribution. Instead the user specifies:

1) the type of information to be used in data partitioning and
2) the irregular data partitioning heuristic to be used.

Language extensions have been designed and implemented to allow users to specify the information needed to produce an irregular distribution. Based on user directives, the compiler produces code that, at runtime, passes the user specified partitioning information to a (user specified) partitioner.

To the best of the authors' knowledge, the implementation described in this paper was the first distributed memory compiler to provide this kind of support. User specified partitioning has recently been implemented in the D System Fortran 77D compiler [16]; the CHAOS runtime support described in this paper has been employed in this implementation. In the Vienna Fortran [45] language definition a user can specify a customized distribution function. The runtime support and compiler transformation strategies described here can also be applied to Vienna Fortran.

These ideas have been implemented using the Syracuse Fortran 90D/HPF compiler [4]. The following assumptions have been made:

1) irregular accesses are carried out in the context of a single or multiple statement parallel loops. In these loops dependence between iterations may occur due to reduction operations only (e.g., addition, max, min, etc.) and
2) irregular array accesses occur as a result of a single level of indirection with a distributed array that is indexed directly by the loop variable.

C. Organization

This paper is organized as follows. The context of the work is outlined in Section II. Section III describes the runtime technique that saves and reuses results from previously performed loop preprocessing. Section IV describes the data structure, the compiler transformations, and the language extensions used to control compiler-linked runtime partitioning. Section V presents the runtime support developed for coupling data partitioners, for partitioning workload and for managing irregular data distributions. Section VI presents data to characterize the methodological performance. Section VII provides a summary of related work, and Section VIII concludes.

II. Overview

A. Problem Partitioning and Application Codes

It is useful to describe application codes to introduce the motivation behind preprocessing. This section first describes two application codes (an unstructured Euler solver and a molecular dynamics code) that consist of a sequence of loops with indirectly accessed arrays; these are loops analogous to those depicted in Fig. 1. This section then describes a combustion code with a regular data access pattern but with highly nonuniform computational costs. In that code, computational costs vary dynamically and cannot be estimated until runtime.

A.1. Codes With Indirectly Accessed Arrays

The first application code is an unstructured Euler solver used to study the flow of air over an airfoil [29], [21]. Complex aerodynamic shapes require high resolution meshes and, consequently, large numbers of mesh points. A mesh vertex is an abstraction represented by Fortran array data structures. Physical values (e.g., velocity, pressure) are associated with each mesh vertex. These values are called *flow variables* and are stored in arrays. Calculations are carried out using loops over the list of edges that define the connectivity of the vertices. For instance, Fig. 1 sweeps over *nedges* mesh edges. Loop iteration *i* carries out a computation involving the edge that connects vertices *edge*1(*i*) and *edge*2(*i*).

To parallelize an unstructured Euler solver, one needs to partition mesh vertices (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 are determined by the mesh generation strategy [42], [28]. Fig. 2 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 an irregular computational mesh are numbered frequently does not have a useful correspondence to the connectivity pattern (edges) of the mesh. Mesh points are partitioned to minimize communication.

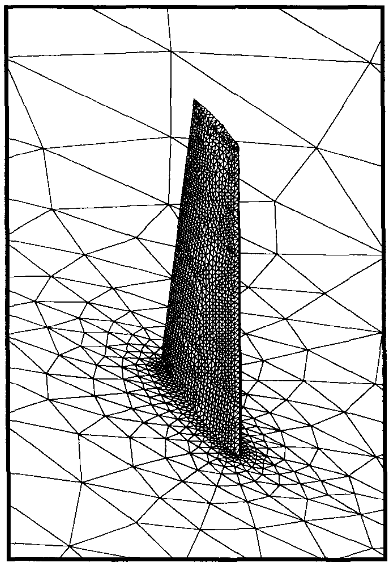


Fig. 2. An example unstructured mesh.

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) estimates of the computational load associated with each mesh point. For instance, a user might choose a partitioner that is based on coordinates [3] to partition data. A coordinate bisection partitioner decomposes data using the spatial location of vertices in the mesh. If the user chooses a graph based partitioner, such as the spectral partitioner [39], the connectivity of the mesh could be used to decompose the data.

The next step in parallelizing this application involves assigning equal amounts of work to processors. An unstructured Euler solver consists of a sequence of loops that sweep over a mesh. Computational work associated with each loop must be partitioned between processors to balance load. The approach used in this paper is to assign all work associated with a given loop iteration to a single processor. Consider a loop that sweeps over mesh edges, closely resembling the loop depicted in Fig. 1. Mesh edges would be partitioned so that 1) good load balance is obtained and 2) computations mostly employ locally stored data.

Other unstructured problems have analogous indirectly accessed arrays. For instance, consider the nonbonded force calculation in the molecular dynamics code CHARMM [5]. Fig. 4 depicts the nonbonded force calculation loop. Force components associated with each atom are stored as Fortran arrays. The outer loop L1 sweeps over all atoms; in this discussion, it is assumed that L1 is a parallel loop. Each iteration of L1 is carried out on a single processor, so loop L2 need not be parallelized.

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

Atoms are partitioned to reduce interprocessor communication in the nonbonded force calculation loop (Fig. 4). Fig. 3 depicts two possible distributions of atoms of a Myoglobin molecule to four processors in which shading is used to represent the assignment of atoms to processors. Data sets associated with sequential versions of CHARMM associate each atom with an arbitrary index number. Fig. 3a shows 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. Fig. 3b depicts a distribution of atoms to processors carried out using an inertial bisection partitioner [3]. Fig. 3b has a much smaller amounts of surface area between the portions of the molecule associated with each processor compared to that of Fig. 3a.

Table I summarizes the application area specific terminology used to describe data array elements, loop iterations, array distributions and loop iteration partition.

A.2. A Code With Time Varying Computational Costs

This section describes a type of application code that is qualitatively different from the unstructured Euler and molecular dynamics codes previously discussed. This type of code is used to carry out detailed time dependent, multidimensional flame simulations. The calculation cycles between two distinct phases. The first phase (*convection*) calculates fluid convection over a Cartesian mesh. The second phase (*reaction*) solves the ordinary differential equations used to represent chemical reactions and energy release. During the reaction phase, a set of local computations are carried out at each mesh point. The computational costs associated with the reaction phase varies from mesh point to mesh point since at each mesh point an adaptive method is used to solve the system of ordinary differential equations. Arrays in this application are not indirectly accessed as in the previous two example applications.

Fig. 5 presents a simplified one dimensional version of this code. The convection phase (loop nest L2) consists of a sweep over a structured mesh involving array elements located at nearest neighbor mesh points. The reaction phase (loop nest L3) involves only local calculations. The computational cost associated with the function *Adaptive_Solver* depends on the value of x(i). It is clear that the cost of *Adaptive_Solver* can vary from mesh point to mesh point. The cost of *Adaptive_Solver* at a given mesh point changes slowly between iterations of the outer loop L1.

There are a number of strategies that can be used in partitioning data and work associated with this flame code. If the convection calculations comprise the bulk of the computation time, it would be reasonable to partition the mesh (arrays x, y, and z in Fig. 5) into equal sized blocks.

However, the reaction calculations (loop nest L3 in Fig. 5) usually comprise at least half of the total computational cost. A majority of the work associated with the reaction phase of the calculation is carried out on a small fraction of the mesh points. The current approach involves maintaining a block mapping of the mesh (arrays x, y, and z) during the convection phase. In order to ensure a good load balance during the reaction phase, only expensive reaction calculations are redistributed. In Fig. 5, array element $x(i)$ must be transmitted in order to redistribute the reaction calculation for mesh point i. Once the reaction calculation is carried out, the solution $z(i)$ is returned to the processor to which it is assigned. At a given mesh point, the cost associated with a reaction calculation generally varies gradually as a problem progresses. This property provides a way to estimate reaction calculation costs in the subsequent computation step.

TABLE I
APPLICATION AREA SPECIFIC TERMINOLOGY

Program Representation	Unstructured Mesh	Molecular Dynamics
Data Array Elements	Physical State for Each Mesh Vertex	Force Components for Each Atom
Loop Iterations	Mesh Edges	Partner List
Array Distribution	Partition of Mesh Vertices	Partition of Atoms
Loop Iteration Partition	Partition of Mesh Edges	Partition of Non-bonded Force Calculations

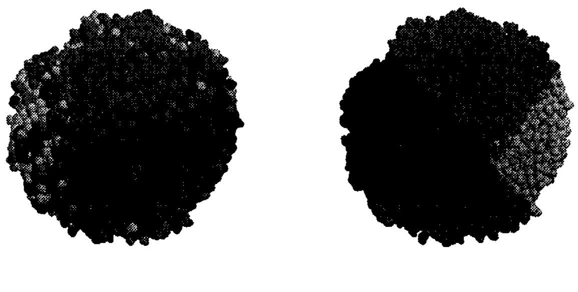


Fig. 3. Distribution of atoms on four processors.

```
L1: do i = 1, NATOMS
    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
```

Fig. 4. Nonbonded force calculation loop from CHARMM.

```
L1: do time = 1, timesteps
  C Convection Phase:
    L2: do i = 1, NPOINTS
            x(i) = x(i) + F(y(i), y(i-1), y(i), y(i+1), z(i))
        end do
        y(1:NPOINTS) = x(1:NPOINTS)
  C Reaction Phase:
    L3: do i = 1, NPOINTS
            z(i) = Adaptive_Solver(x(i))
        end do
    end do
```

Fig. 5. Overview—combustion code.

B. Solving Irregular Problems

This section describes how irregular problems can be solved efficiently on distributed memory machines. On distributed memory machines the data and the computational work must be divided between individual processors. The criteria for partitioning are minimizing the volume of interprocessor data communication and good load-balancing.

Once distributed arrays have been partitioned, each processor ends up with a set of globally indexed distributed array elements. Each element in a size N distributed array, A, is assigned to a particular home processor. In order for other processors to be able to access a given element $A(i)$ of the distributed array, the home processor and local address of $A(i)$ must be determined. A *translation table* is built that lists the home processor and the local address for each array element.

Memory considerations make it clear that it is not always feasible to place a copy of the translation table on each processor, so the translation table must be distributed between processors. This is accomplished by distributing the the translation table in blocks, i.e., putting the first N/P elements on the first processor, the second N/P elements on the second processor, etc., where P is the number of processors. When an element $A(m)$ of distributed array A is accessed, the home processor and local offset are found in the portion of the distributed translation table stored in processor $\lfloor((m-1)/N) \times P\rfloor + 1$. The translation table lookup aimed at discovering the home processor and the offset associated with a global distributed array index is called a *dereference request*.

Consider the irregular loop L2 in Fig. 1 that sweeps over the edges of a mesh. In this case, distributing data arrays x and y corresponds to partitioning the mesh vertices; partitioning loop iterations corresponds to partitioning edges of the mesh. Hence, each processor gets a subset of loop iterations (edges). An edge *i* that has both end points (*edge*1(*i*) and *edge*2(*i*)) inside the same partition (*processor*) requires no outside information. On the other hand, edges which cross partition boundaries require data from other processors. Before executing the computation for such an edge, a processor must retrieve the required data from other processors.

There is typically a nontrivial communication latency, or message startup cost, in distributed memory machines. Communication is vectorized to reduce the effect of communication latency and software caching is carried out to reduce communication volume. To carry out either optimization, it is extremely helpful to have a priori knowledge of data access patterns. In irregular problems, it is generally not possible to predict data access patterns at compile time. For example, the values of indirection arrays edge1 and edge2 of loop L2 in Fig. 1 are known only at runtime because they depend on the input mesh. During program execution, preprocessing examines the data references of distributed arrays. Each processor precomputes which data need to be exchanged. The result of this preprocessing is a *communication schedule* [30].

Each processor uses communication schedules to exchange required data before and after executing a loop. The same schedules can be used repeatedly, as long as the data reference patterns remain unchanged. In Fig. 1, loop L2 is carried out many times inside loop L1. As long as the indirection arrays edge1 and edge2 are not modified within L1, it is possible to reuse communication schedules for L2. Schedule reuse will be discussed in detail in Section III.

C. Communication Vectorization and Software Caching

The process of generating and using schedules to carry out communication vectorization and software caching can be described with the help of the example shown in Fig. 1. The arrays x, y, edge1, and edge2 are partitioned between the processors of the distributed memory machine. Assume that arrays x and y are distributed in the same fashion. Array distributions are stored in a distributed translation table. These local indirection arrays are passed to the procedure *localize* as shown in statement S1 in Fig. 6.

Fig. 6 contains the preprocessing code for the simple irregular loop L2 shown in Fig. 1. In this loop, values of array y are updated using the values stored in array x. Hence, a processor may need an off-processor array element of x to update an element of y and it may update an off-processor array element of y. The goal is to compute 1) a *gather schedule*—a communication schedule that can be used for fetching off-processor elements of x, and 2) a *scatter schedule*—a communication schedule that can be used to send updated off-processor elements of y. However, the arrays x and y are referenced in an identical fashion in each iteration of the loop L2, so a single schedule that represents data references of either x or y can be used for fetching off-processor elements of x and sending off-processor elements of y.

```
 C  Create the required schedules (Inspector)
S1  Collect indirection array traces and call CHAOS procedure localize to compute schedule
 C  The actual computation (Executor)
S2  call zero_out_buffer(x(begin_buffer), off_proc)
S3  call gather(x(begin_buffer), x, schedule)
S4  do i=1, n_local_edges
S5    y(local_edge1(i)) = y(local_edge1(i)) + f(x(local_edge1(i)), x(local_edge2(i)))
S6    y(local_edge2(i)) = y(local_edge2(i)) + g(x(local_edge1(i)), x(local_edge2(i)))
S7  end do
S8  call scatter_add(y(begin_buffer), y, schedule)
```

Fig. 6. Node code for simple irregular loop.

A sketch of how the procedure *localize* works is shown in Fig. 7. The globally indexed reference pattern used to access arrays x and y is stored in the array part_edge. The procedure *localize* dereferences and translates part_edge so that valid references are generated when the loop is executed. The buffer for each data array immediately follows the on-processor data for that array. For example, the buffer for data array y begins at y(begin_buffer). Hence, when *localize* translates part_edge to local_edge, the off-processor references are modified to point to buffer addresses. The procedure *localize* uses a hash table to remove any duplicate references to off-processor elements so that only a single copy of each off-processor datum is transmitted. When the off-processor data are collected into the buffer using the schedule returned by *localize*, the data are stored in a way such that execution of the loop using the local_edge accesses the correct data.

The executor code starting at S2 in Fig. 6 carries out the actual loop computation. In this computation the values stored in the array y are updated using the values stored in x. During the computation, accumulations to off-processor locations of array y are carried out in the buffer associated with array y. This makes it necessary to initialize the buffer corresponding to off-processor references of y. To perform this action, the function *zero_out_buffer* shown in statement S2 is called. After the loop computation, the data in the buffer location of ar-

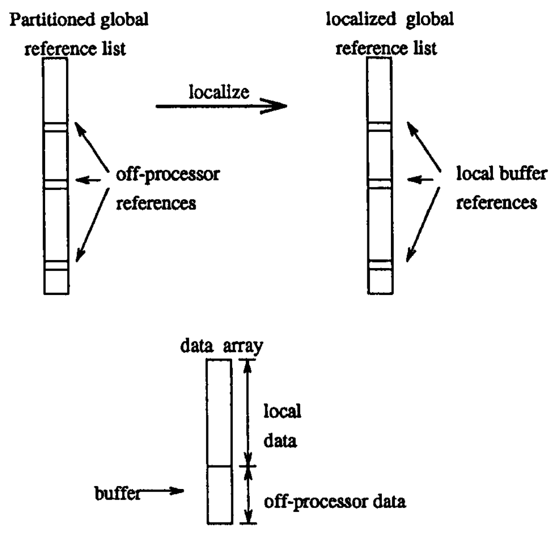


Fig. 7. Index translation by localize mechanism.

ray y are communicated to the home processors of these data elements (*scatter_add*). There are two potential communication points in the executor code, i.e., the *gather* and the *scatter_add* calls. The *gather* on each processor fetches all the necessary x references that reside off-processor. The *scatter_add* call accumulates the off-processor y values. A detailed description of the functionality of these procedures is given in Ponnusamy [33].

D. Overview of CHAOS

Efficient runtime support has been developed to deal with problems that consist of a sequence of clearly demarcated concurrent computational phases. The project is called CHAOS; the runtime support is called the CHAOS library [33]. The CHAOS library is a superset of the PARTI library [30], [38], [11].

Solving concurrent irregular problems on distributed memory machines using CHAOS runtime support involves five major steps (Fig. 8). The first three steps in the figure concern mapping data and computations onto processors. This section provides a brief description of these steps and will discuss them in detail in later sections.

Initially, the distributed arrays are decomposed into a known regular manner.

1) The first step is to decompose the distributed array irregularly with the user provided information. When the user chooses connectivity as a piece of information to be used for data partitioning, preprocessing is required to generate GeoCoL graph (see Section V) before information can be passed to a partitioner. In Phase A of Fig. 8, CHAOS procedures can be called to do the necessary preprocessing. For example, the user may employ a partitioner that uses the connectivity of the mesh shown in Fig. 2 or may use a partitioner that uses the spatial information of the mesh vertices. The partitioner calculates how data arrays should be distributed.
2) In Phase B, the newly calculated array distributions are

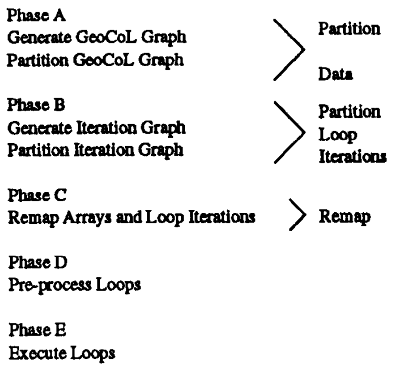


Fig. 8. Solving irregular problems.

used to decide how loop iterations are to be partitioned among processors. This calculation takes into account the processor assignment of the distributed array elements accessed in each iteration. A loop iteration is assigned to the processor that has the maximum number of local distributed arrays elements accessed in that iteration. Once data are distributed, based on the access patterns of each iteration and data distribution, the runtime routines for this step determine on which processor each iteration will be executed.
3) Once new data and loop iteration distributions are determined, Phase C carries out the actual remapping of arrays from the old distribution to the new distribution.
4) In Phase D, the preprocessing needed for software caching, communication vectorization and index translation is carried out. In this phase, communication schedules are generated that can be used to exchange data among processors.
5) Finally, in Phase E, information from the earlier phases is used to carry out the computation and communication.

CHAOS and PARTI procedures have been used in a variety of applications, including sparse matrix linear solvers, adaptive computational fluid dynamics codes, molecular dynamics codes, and a prototype compiler [38] aimed at distributed memory multiprocessors.

E. Overview of Existing Language Support

While these data decomposition directives are presented in the context of Fortran D, the same optimizations and analogous language extensions could be used for a wide range of languages and compilers such as Vienna Fortran, pC++, and HPF. Vienna Fortran, Fortran D, and HPF provide a rich set of data decomposition specifications. A definition of such language extensions may be found in Fox et al. [14], Loveman et al. [13], and Chapman et al. [7], [8]. Fortran D and HPF require that users explicitly define how data are to be distributed. Vienna Fortran allows users to write procedures to generate user defined distributions. The techniques described in this paper are being adapted to implement user defined distributions in the Vienna Fortran compiler; details of the Vienna Fortran based work will be reported elsewhere.

Fortran D and Vienna Fortran can be used to *explicitly* specify an irregular partition of distributed array elements. Fig.

9 presents an example of such a Fortran D declaration. In Fortran D, one declares a template called a *distribution* that is 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 users 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 statement S3, of Fig. 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*.

The difficulty with the declarations depicted in Fig. 9 is that *it is not obvious how to partition the irregularly distributed array.* The *map* array that gives the distribution pattern of `irreg` has to be generated separately by running a partitioner (the user may supply the partitioner or use one from a library). The Fortran D constructs are not rich enough for the user to couple the generation of the `map` array to the program compilation process. While there are a wealth of partitioning heuristics available, coding such partitioners from scratch can represent a significant effort. There is also no standard interface between the partitioners and the application codes. Section IV will discuss language extensions and compiler support to interface data partitioners.

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

Fig. 9. Fortran D irregular distribution.

Fig. 10 shows an irregular Fortran 90D Forall loop that is equivalent to the sequential loop L2 in Fig. 1. The loop L1 represents a sweep over the edges of an unstructured mesh. Since the mesh is unstructured, an indirection array has to be used to access the vertices during a loop over the edges. In loop L1, a sweep is carried out over the edges of the mesh and the reference pattern is specified by integer arrays `edge1` and `edge2`. Loop L1 carries out reduction operations. That is, the only type of dependency between different iterations of the loop is the one in which they may produce a value to be accumulated (using an associative and commutative operation) in the same array element. Fig. 2 shows an example of an unstructured mesh over which such computations will be carried out. For example, the loop L1 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). Clearly, each vertex of the mesh is updated as many times as the number of neighboring vertices.

The definition of the Forall construct in HPF follows copy-in-copy-out semantics—loop carried dependencies are not allowed. This implementation allows loop carried dependencies that arise due to reduction operations. The reduction operations in a Forall construct are specified using the Fortran D REDUCE construct. Reduction inside a Forall construct is important for representing a considerable set of scientific computations such as those found in sparse and unstructured problems [9]. This representation also preserves explicit parallelism available in the underlying computations.

```
C Sweep over edges: Loop L1
   FORALL i = 1, nedges
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
```

Fig. 10. Example irregular loop in Fortran D.

III. Communication Schedule Reuse

The cost of carrying out an inspector (phases B, C, and D in Fig. 8) can be amortized when the information produced by the inspector is computed once and then used repeatedly. The compile time analysis needed to reuse inspector communication schedules is touched upon in Das et al. [12].

This paper proposes a conservative method that in many cases allows reuse of the results from inspectors. The results from an inspector for loop L can be reused as long as:

- the distributions of data arrays referenced in loop L have remained unchanged since the last time the inspector was invoked,
- there is no possibility that the indirection arrays associated with loop L have been modified since the last inspector invocation, and
- the loop bounds of L have not changed.

The compiler generates code that, at runtime, maintains a record of when the statements or array intrinsics of a Fortran 90D loop may have written to a distributed array that is used to indirectly reference another distributed array. In this scheme, before executing a loop, this runtime record is checked to see whether any indirection arrays may have been modified since the last time the loop was invoked.

In this presentation it is assumed that an inspector is being carried out for a Forall loop. Also assumed is that all indirect array references to any distributed array y are of the form *y*(*ia*(*i*)) where ia is a distributed array and *i* is a loop index associated with the Forall loop.

The information about an array is stored in a runtime data structure called *data access descriptor* (*DAD*). A DAD for a distributed array contains (among other things) the current distribution type of each dimension of the array (e.g., block, cyclic) and the size of the array. In order to generate correct distributed memory code, whenever the compiler generates code that references a distributed array, the compiler must have access to the array's DAD. In this scheme, a *global* data structure is also maintained to keep track of modifications of *any array* with a given DAD.

A global variable n_mod is maintained that represents the cumulative number of Fortran 90D loops, array intrinsics or statements that have modified any distributed array. Note that this scheme does not count the number of assignments to the distributed array, instead it counts the number of times the program has executed any block of code that writes to a distributed array.[1] The variable n_mod may be viewed as a global time stamp. Each time an array A with a given data access descriptor DAD(A) is modified, a global data structure last_mod is updated to associate DAD(A) with the current value of the global variable n_mod (i.e., the current global time stamp). Thus when a loop, array intrinsic, or statement modifies A, last_mod(DAD(A)) is set to n_mod. If the array A is remapped, it means that DAD(A) changes. In this case, n_mod is incremented and then last_mod(DAD(A)) is set to n_mod.

The first time a Forall loop L is executed, inspector for that loop is carried out. Assume that L has *m* data arrays $x^i_L, 1 \le i \le m$, and *n* indirection arrays, $ind^j_L, 1 \le j \le n$. Each time an inspector for L is carried out, the following information is stored:

1) $\mathrm{DAD}(x^i_L)$ for each unique data array x^i_L, for $1 \le i \le m$,
2) $\mathrm{DAD}(ind^j_L)$ for each unique indirection array ind^j_L, for $1 \le j \le n$,
3) $\mathrm{last\_mod}(\mathrm{DAD}(ind^j_L))$, for $1 \le j \le n$, and
4) the loop bounds of L.

The values of $\mathrm{DAD}(x^i_L)$, $\mathrm{DAD}(ind^j_L)$, and $\mathrm{last\_mod}(\mathrm{DAD}(ind^j_L))$ stored by L's inspector are designated as $\mathrm{L.DAD}(x^i_L)$, $\mathrm{L.DAD}(ind^j_L)$, and $\mathrm{L.last\_mod}(\mathrm{DAD}(ind^j_L))$, respectively.

For a given data array x^i_L and an indirection array ind^j_L in a Forall loop L, *two sets of data access descriptors* are maintained. For instance,

1) $\mathrm{DAD}(x^i_L)$, the current global data access descriptor associated with x^i_L and
2) $\mathrm{L.DAD}(x^i_L)$, a record of the data access descriptor that was associated with x^i_L when L carried out its previous inspector

are maintained. Each indirection array ind^j_L also maintains two time stamps:

- $\mathrm{last\_mod}(\mathrm{DAD}(ind^j_L))$ is the global time stamp associated with the current data access descriptor of ind^j_L and
- $\mathrm{L.last\_mod}(\mathrm{DAD}(ind^j_L))$ is the global time stamp of data access descriptor $\mathrm{DAD}(ind^j_L)$, last recorded by L's inspector.

The first time L is executed, L's inspector is carried out, the following checks are performed before subsequent executions of L. If any of the following conditions are not met, the inspector must be repeated for L:

1) $\mathrm{DAD}(x^i_L).\mathrm{EQU.\ L.DAD}(x^i_L),\ 1 \le i \le m,$
2) $\mathrm{DAD}(ind^j_L).\mathrm{EQU.\ L.DAD}(ind^j_L),\ 1 \le j \le n,$
3) $\mathrm{last\_mod}(\mathrm{DAD}(ind^j_L))\ .\mathrm{EQU.\ L.last\_mod}(\mathrm{L.DAD}(ind^j_L)),\ 1 \le j \le n$, and
4) the loop bounds of L remain unchanged.

As the above algorithm tracks possible array modifications at runtime, there is potential for high runtime overhead in some cases. The overhead is likely to be small in most computationally intensive data parallel Fortran 90 codes (see Section VI). Calculations in such codes primarily occur in loops or Fortran 90 array intrinsics, so so it is necessary to record modifications to a DAD *once* per loop or array intrinsic call.

The same method is employed to track possible changes to arrays used in the construction of the data structure produced at runtime to link partitioners with programs. This data structure is called a *GeoCoL* graph, and it will be described in Section IV.A.1. This approach makes it simple for a compiler to avoid generating a new GeoCoL graph and carrying out a potentially expensive data repartition when no change has occurred.

IV. Coupling Partitioners

In irregular problems, it is often desirable to allocate computational work to processors by assigning all computations that involve a given loop iteration to a single processor [38]. Consequently, *both* distributed arrays and loop iterations are partitioned using a two-phase approach (Fig. 8). In the first

1. Note that a Forall construct or an array construct is an atomic operation from the perspective of language semantics, and therefore, it is sufficient to consider one write per construct rather than one write per element.

phase, termed the *data partitioning* phase, distributed arrays are partitioned. In the second phase, called *loop iteration partitioning*, loop iterations are partitioned using the information from the first phase. This appears to be a practical approach, as in many cases the same set of distributed arrays are used by many loops. The following two subsections describe the phases.

A. Data Partitioning

When distributed arrays are partitioned, loop iterations have not yet been assigned to processors. Assume that loop iterations will be partitioned using a user-defined criterion similar to that used for data partitioning. In the absence of such a criterion, a compiler will choose a loop iteration partitioning scheme, e.g., partitioning loops so as to minimize nonlocal distributed array references. This approach makes an implicit assumption that most (although not necessarily all) computation will be carried out in the processor associated with the variable appearing on the left-hand side of each statement—this approach is called the *almost owner computes rule* [36].

There are many partitioning heuristics methods available based on physical phenomena and proximity [39], [3], [44], [17]. Table II lists some of the commonly used heuristics and the types of information they use for partitioning. Most data partitioners make use of undirected connectivity graphs and spatial information. Currently these partitioners must be coupled to user programs manually. This manual coupling is particularly troublesome and tedious when users wish to make use of parallelized partitioners. Further, partitioners use different data structures and are very problem dependent, making it extremely difficult to adapt to different (but similar) problems and systems.

TABLE II
COMMON PARTITIONING HEURISTICS

Partitioner	Reference	Spatial Information	Connectivity Information	Vertex Weight	Edge Weight
Spectral Bisection	[39]		✓	✓	✓
Coordinate Bisection	[3]	✓		✓	
Hierarchical Decomposition	[10]	✓		✓	
Simulated Annealing	[27]		✓	✓	✓
Neural Network	[27]		✓	✓	✓
Genetic Algorithms	[27]		✓	✓	✓
Inertial Bisection	[31]	✓		✓	
Kernighan - Lin	[22]		✓	✓	✓

A.1. Interface Data Structures for Partitioners

Partitioners are linked to programs by using a data structure that stores information on which data partitioning is to be based. Data partitioners can make use of different kinds of program information. Some partitioners operate on data structures that represent undirected graphs [39], [22], [27]. Graph vertices represent array indices; graph edges represent dependencies. Consider the example loop L1 in Fig. 10. The graph vertices represent the N elements of arrays x and y. The graph edges of the loop in Fig. 10 are the union of the edges linking vertices $edge1(i)$ and $edge2(i)$.

In some cases, it is possible to associate geometrical information with a problem. For instance, meshes often arise from finite element or finite difference discretizations. In such cases, each mesh point is associated with a location in space. Each graph vertex can be assigned a set of coordinates that describe its spatial location. These spatial locations can be used to partition data structures [3], [31].

Vertices may also be assigned weights to represent estimated computational costs. In order to accurately estimate the computational costs, partitioners need information on how work will be partitioned. One way of deriving weights is to make the implicit assumption that an owner computes rule will be used to partition work. Under this assumption, computational cost associated with executing a statement will be attributed to the processor owning a left-hand side array reference. The weight associated with a vertex in the loop L2 of Fig. 10 would be proportional to the degree of the vertex, assuming functions f and g have identical computational costs. Vertex weights can be used as the sole partitioning criterion in problems in which computational costs dominate. Examples of such code include the flame simulation code described in Section II.A.2 and "embarrassingly parallel problems" [9], where computational cost predominates.

A given partitioner can make use of a combination of connectivity, geometrical, and weight information. For instance, sometimes it is important to take estimated computational costs into account when carrying out coordinate or inertial bisection for problems where computational costs vary greatly from node to node. Other partitioners make use of both geometrical and connectivity information [10].

Since the data structure that stores information on which data partitioning is to be based can represent Geometrical, Connectivity and/or Load information, it is called the GeoCoL data structure.

More formally, a GeoCoL graph $G = (V, E, W_v, W_e, C)$ consists of

1) a set of vertices $V = \{v_1, v_2, ..., v_n\}$, where $n = |V|$,
2) a set of undirected edges $E = \{e_1, e_2, ..., e_m\}$, where $m = |E|$,
3) a set of vertex weights $W_v = \{W_v^1, W_v^2, \ldots, W_v^n\}$,
4) a set of edge weights $W_e = \{W_e^1, W_e^2, \ldots, W_e^m\}$, and
5) a set of coordinate information, for each vertex, of dimension d, $C = \{< c_1^1, \ldots, c_d^1 >, \ldots, < c_1^n, \ldots, c_d^n >\}$.

A.2. Generating the GeoCoL Data Structure via a Compiler

This section proposes an executable directive CONSTRUCT that can be employed to direct a compiler to generate the GeoCoL data structures. A user can specify spatial information using the keyword GEOMETRY.

The following is an example of a GeoCoL declaration that specifies geometrical information:

C$ CONSTRUCT G1 (N, GEOMETRY(3, xcord, ycord, zcord)).

This statement defines a GeoCoL data structure called G1 having *N* vertices with spatial coordinate information specified by arrays xcord, ycord, and zcord. The GEOMETRY construct is closely related to the geometrical partitioning or *value-based decomposition* directives proposed by von Hanxleden [16].

Similarly, a GeoCoL data structure that specifies only vertex weights can be constructed using the keyword LOAD as follows.

C$ CONSTRUCT G2 (N, LOAD(weight)).

Here, a GeoCoL structure called G2 consists of *N* vertices with vertex *i* having LOAD *weight(i)*.

The following example illustrates how connectivity information is specified in a GeoCoL declaration. The integer arrays n1 and n2 list the vertices associated with each of E graph edges and integer arrays n1 and n3 list vertices for another set of E edges.

C$ CONSTRUCT G3 (N, LINK(E, n1, n2), LINK(E, n1, n3)).

The keyword LINK is used to specify the edges associated with the GeoCoL graph. The resultant edges of the GeoCoL data structure are the union of 1) edges linking *n*1(*i*) and *n*2(*i*) and 2) edges linking *n*1(*i*) and *n*3(*i*).

Any combination of spatial, load, and connectivity information can be used to generate the GeoCoL data structures. For instance, the GeoCoL data structure for a partitioner that uses both geometrical and connectivity information can be specified as follows:

C$ CONSTRUCT G4 (N, GEOMETRY
(3, xcord, ycord, zcord), LINK(E, edge1, edge2)).

Once the GeoCoL data structure is constructed, data partitioning is carried out. Assume that there are P processors. At compile time *dependency coupling code* is generated. This code generates calls to the runtime support that, when the program executes:

1) generates the GeoCoL data structure,
2) passes the GeoCoL data structure to a *data partitioning* procedure where the partitioner partitions the GeoCoL into P subgraphs, and
3) passes the new distribution information (the assignment of GeoCoL vertices to processors) to a runtime procedure to redistribute data.

The GeoCoL data structure is constructed from the initial default distribution of the distributed arrays. Once the partitioner generates a new distribution, the arrays can be redistributed based on it. A communication schedule is built and used to redistribute the arrays from the default to the new distribution.

Vienna Fortran [45] provides support for the user to specify a function for distributing data. Within the function, the user can perform any processing to specify the data distribution.

B. Examples of Linking Data Partitioners

Fig. 11 illustrates a possible set of partitioner coupling directives for the loop L1 in Fig. 10. Statements S1 to S4 produce a default initial distribution of data arrays x and y and the indirection arrays edge1 and edge2 in loop L2. The statements S5 and S6 direct the generation of code to construct the GeoCoL graph and call the partitioner. Statement S5 indicates that the GeoCoL graph edges are to be generated based on the indirection arrays edge1 and edge2. This information is provided by using the keyword LINK in the CONSTRUCT directive. The motivation for using the indirection arrays to construct the edges is that they represent the underlying data access patterns of the arrays x and y in loop L1. When the GeoCoL graph with edges representing the data access pattern is passed to the partitioner, the partitioner tries to break the graph into subgraphs such that the number of edges cut between the subgraphs is minimal. Hence, communication between processors is minimized. The statement S6 in the figure calls the recursive spectral bisection (RSB) partitioner with GeoCoL as input. The user is provided with a library of commonly available partitioners and can choose among them. Also, the user can link a customized partitioner as long as the calling sequence matches that of the partitioners in the library. Finally, the distributed arrays are remapped in statement S7 using the new distribution returned by the partitioner.

```
   REAL*8 x(nnodes), y(nnodes)
   INTEGER edge1(nedges), edge2(nedges)
S1 DYNAMIC, DECOMPOSITION reg(nnodes),reg2(nedges)
S2 DISTRIBUTE reg(BLOCK), reg2(BLOCK)
S3 ALIGN x, y with reg
S4 ALIGN edge1, edge2 with reg2
   ....
   call read_data(edge1, edge2, ...)
S5 CONSTRUCT  G(nnodes),LINK(nedges,edge1, edge2))
S6 SET distfmt BY PARTITIONING G USING RSB
S7 REDISTRIBUTE reg(distfmt)
C  Loop over edges involving x, y
L2 FORALL i = 1, nedges
     REDUCE (SUM, y(edge1(i)), f(x(edge1(i)), x(edge2(i))))
     REDUCE (SUM, y(edge2(i)), g(x(edge1(i)), x(edge2(i))))
   END FORALL
   ....
```

Fig. 11. Example of implicit mapping in Fortran 90D.

Fig. 12 illustrates code similar to that shown in Fig. 11 except that the use of geometric information is shown. Arrays xc, yc, and zc, which carry the spatial coordinates for elements in x and y, are aligned with the same decomposition to which arrays x and y are aligned. Statement S5' specifies that the GeoCoL data structure is to be constructed using geometric information. S6' specifies that recursive coordinate bisection (RCB) partitioner is used to partition the data.

Recall from Section II.A.2 that the computation in the combustion code cycles over a convection phase and a reaction phase. The data access pattern in the convection phase in-

```
S5' CONSTRUCT G (nnodes, GEOMETRY(3, xc, yc, zc))
S6' SET distfmt BY PARTITIONING G USING RCB
S7' REDISTRIBUTE reg(distfmt)
```

Fig. 12. Example of implicit mapping using geometry information in Fortran 90D.

```
S1  DYNAMIC, DECOMPOSITION grid(NPOINTS)
S2  DISTRIBUTE grid(BLOCK)
S3  ALIGN x(:), y(:), z(:), wt(:) WITH grid(BLOCK)
K1  wt(1:NPOINTS) = 1
K2  do J = 1, n_timesteps
C     Phase 1: Navier Stokes Solver - Convection Phase - BLOCK data distribution
K3  FORALL i = 1, NPOINTS
K4    x(i) = x(i) + F(y(i), y(i-1), y(i), y(i+1), z(i))
K5  END FORALL
S4  CONSTRUCT G (NPOINTS, LOAD(wt))
S5  SET mydist BY PARTITIONING G USING BIN_PACKING
S6  REDISTRIBUTE grid(mydist)
C     Phase 2: Adaptive ODE Solver - Reaction Phase - IRREGULAR data distribution
K6  wt(1:NPOINTS) = 1
K7  FORALL i = 1, NPOINTS
K8    z(i) = Adaptive_Solver(x(i),wt(i))
K9  END FORALL
S7  REDISTRIBUTE grid(BLOCK)
K10 end do
```

Fig. 13. An example of adaptive partitioning using Fortran 90D.

volves access to only nearest neighbor array elements. Hence, during the convection phases it is reasonable to make use of a BLOCK distribution of data for arrays x, y, and z. Statements S1 through S3 in Fig. 13 produce BLOCK distribution of data arrays. In the reaction phase, the amount of work done at each mesh point varies as time progresses, and no communication occurs. The computational cost of the reaction phase at each mesh point in the current time step is stored in array wt. This cost information is used to distribute data arrays in the reaction phase of the next time step. A bin-packing heuristic is invoked to obtain the data distribution for the reaction phase. The statements S4 through S6 carry out the data distribution for the reaction phase.

C. Loop Iteration Partitioning

Once data have been partitioned, computational work can 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 within loops may be needed, 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 ib(i), OWNER(ib(i)), while statement S2 would be computed on the owner of ia(i), OWNER(ia(i)). The value of y(ib(i)) would have to be communicated whenever OWNER(ib(i)) ≠ OWNER(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, the above loop could be specified 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, OWNER(x(i)), where the sizes of arrays ia and ib are equal to the number of iterations. Similar capabilities exist in Vienna Fortran.

When an ON clause is not explicitly specified, it is the compiler's responsibility to determine where to compute each iteration. An alternate policy to the owner computes rule is to assign all work associated with a loop iteration to a given processor. The current default is to employ a scheme that executes a loop iteration on the processor that is the home of the largest number of distributed array references in an iteration. This scheme is referred to as the "almost owner computes rule."

V. Runtime Support

This section briefly discusses the functionality of the runtime primitives that are used to perform the steps outlined in Fig. 8. It should be noted that one of the important features of the approach taken in this work is the reliance upon an efficient runtime system.

The runtime support for compiler-embedded mapping presented in this paper can be broadly divided into three categories: 1) general support for communication and distributed data management, 2) data partitioning, and 3) iteration partitioning (work assignment). The following subsections briefly describe these primitives.

A. Data Partitioning

The runtime support associated with data partitioning includes procedures for generating the GeoCoL data structure for partitioners (that operate on the GeoCoL data structure) to determine a data distribution, and procedures for remapping data as specified by the partitioner output.

The data structures describing the problem domain are specified by the CONSTRUCT directive discussed earlier. Processing this primitive requires generating a weighted interaction graph representing the computation load and/or communication dependencies. For example, the connectivity edges of the GeoCoL graph might reflect the read/write access patterns of the specified computation.

When connectivity information for the GeoCoL data structure is provided in the form of arrays (e.g., indirection arrays in an irregular loop), preprocessing is required to construct the connectivity graph. The procedures *eliminate_dup_edges* and *generate_geocol* could be used to do the preprocessing. Given the data access pattern information in the form of integer arrays n1 and n2, the GeoCoL graph is constructed by adding an undirected edge $< n1(i), n2(i) >$ between nodes $n1(i)$ and $n2(i)$ of the graph.

Fig. 14 shows the parallel generation of connectivity information in the GeoCoL data structure when integer indirection arrays are provided. Each processor generates the local GeoCoL data structure using the local set of indirection arrays. The local graph is generated by the procedure *eliminate_dup_edges*. For clarity, the local GeoCoL is shown as an adjacency matrix. The local graphs are then merged to form a global distributed graph using the procedure *generate_geocol*. During the merge, if the local graph is viewed as an adjacency matrix stored in compressed sparse row format, processor P_0 collects all entries from the first N/P rows in the matrix from all other processors, where N is the number of nodes (array size) and P is the number of processors. Processor P_1 collects the next N/P rows of the matrix and so on. Since entries for each row may come from many processors there may be duplicate entries. Processors remove duplicate entries when they collect adjacency list entries. The output of procedure *generate_geocol* is a GeoCoL data structure with the global connectivity information.

Any appropriate data partitioner may be used to compute the new data distribution using the GeoCoL graph. Table II lists many of the candidate partitioners for determining the data partitioning. In fact, a user may use any partitioner as long as the input and output data structures conform to those required by other primitives. The output of the partitioner describes a mapping of the data satisfying the desired criteria for load balance and communication minimization.

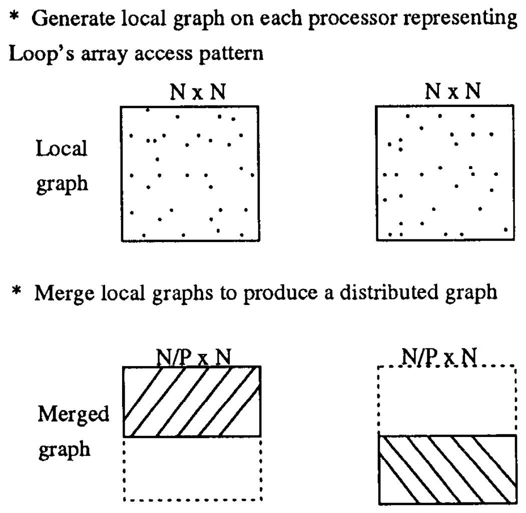

* Generate local graph on each processor representing Loop's array access pattern
N x N
N x N
Local graph
* Merge local graphs to produce a distributed graph
N/P x N
N/P x N
Merged graph

Fig. 14. Parallel generation of GeoCoL graph.

B. Workload Partitioning

Once data are partitioned, computation also must be partitioned. Workload (computation) partitioning refers to determining which processor will evaluate which expressions. Computation 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 seem expensive since, in the first case, the amount of preprocessing overhead can be very high, whereas in the second case communication cost can be very high. This paper has taken an approach which represents a compromise. Each loop iteration is considered individually before assigning it to a processor.

For this purpose, data structures and runtime procedures have been developed to support iteration partitioning. To partition loop iterations, CHAOS uses a graph called the *runtime iteration graph*, or RIG. The RIG associates with each loop iteration *i*, all indices of each distributed array accessed during iteration *i*. A RIG is generated for every loop that references at least one irregularly distributed array.

Using the RIG, for each iteration a list containing the number of distinct data references is computed on each processor. Primitive *deref_rig* uses the RIG and the distributed translation tables to find the processor assignments associated with each distributed array reference. Subsequently, primitive *iteration_partitioner* uses this information to partition iterations. 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.

Note that just as there are many possible strategies that can be used to partition data, there are also many strategies that can be used to partition loop iterations. Currently several techniques have been investigated to specify "workload partitioners" or "iteration partitioners" in which a user can provide a customized heuristic.

C. Data Redistribution

For efficiency in scientific programs, distributions of distributed 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 the problem. This change might result in a poor load balance among processors. Hence, data must be redistributed periodically to maintain this balance.

To obtain an irregular data distribution for an irregular concurrent problem, data arrays are initially partitioned in a known distribution. Then, a heuristic method is applied to obtain an irregular distribution δ_B. Once the new data distribution is obtained, all data arrays associated with distribution δ_A must be transformed to distribution δ_B For example, in solving the Euler equations of an unstructured grid, the flow variables are distributed in this method. Similarly, the loop iterations and the indirection arrays associated with the loop must also be remapped.

To redistribute data and loop iteration space, 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 new distributions.

VI. Experimental Results

This section presents the experimental results for the various techniques presented in this paper for compiler and runtime support for irregular problems. All measurements are performed on the Intel iPSC/860. In particular, this section presents the performance improvements obtained by employing communication schedule reuse, comparing the performance of compiler generated code with that of hand coded versions, and also presents data on the performance of compiler-embedded mapping using various partitioners.

A. Communication Schedule Reuse

This section presents performance data for the schedule saving technique proposed in Section III for the Fortran 90D/HPF compiler implementation.

These performance measurements are for a loop over edges from a 3D unstructured Euler solver [29] for both 10K and 53K mesh points, and for an electrostatic force calculation loop in a molecular dynamics code for a 648 atom water simulation [5]. The functionality of these loops is equivalent to the loop L1 in Fig. 10.

Table III presents the performance results of the compiler generated code with and without the schedule reuse technique. The table presents the execution times of the loops for 100 iterations with distributed arrays decomposed irregularly using a recursive coordinate bisection partitioner. Clearly, being able to reuse communication schedules improves performance significantly. This is because without reuse, schedules must be regenerated at each time step, and therefore, the cost is proportional to the number of iterations.

B. Performance of the Mapper Coupler

This section presents performance results that compare the the costs incurred by the compiler generated mapper coupler procedures with the cost of a hand embedded partitioner.

To map arrays, two different kinds of parallel partitioners are employed: 1) geometry based partitioners (coordinate bisection [3] and inertial bisection [31]) and 2) a connectivity based partitioner (recursive spectral bisection [39]). The performance of the compiler embedded mapper and a hand parallelized version are shown in Tables IV and V.

In Tables IV and V, *Partitioner* 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 & Remap* shows the time taken to build the communication schedule and redistribute data to the new distribution.

Table IV presents the performance of results of the Euler loop with the compiler-linked recursive coordinate bisection partitioner and the BLOCK distribution for a 53K mesh template on 32 processors. Two important observations can be made from Table IV. 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 overhead is partly due to bookkeeping done to reuse schedules and partly due to runtime calculation of loop bounds. Second, the performance of the code using the partitioner is much better than the performance of the block partitioned code even when the cost of executing the partitioner is included.

Table V shows the performance of compiler generated code when two additional partitioners are used; namely, recursive spectral bisection (RSB) and inertial bisection. In Table V, *Partitioner* depicts the time needed to partition the GeoCoL graph data structure using a parallelized version of Simon's single level spectral partitioner [39]. Only a modest effort was made to produce an efficient parallel implementation of the partitioner and it is believed that the performance and the execution time of the partitioner can be tremendously improved by using a multilevel version of the partitioner [2], [18]. The GeoCoL graph is partitioned into a number of subgraphs equal to the number of processors employed. It should be noted that any parallelized partitioner could be used. The *Graph Generation* time depicts the time required to generate the GeoCoL graph.

Clearly, different partitioners perform differently in terms of execution time and quality of load balancing. The best load balancing is obtained by using RSB because the time for the executor phase is minimized. However, the cost of partitioning using RSB is quite high. Thus, the choice of a partitioner should depend on how long the solution of a problem is likely to take (the number of time steps).

Table VI shows the performance of the compiler generated code for the Euler and the molecular dynamics loops on various numbers of processors. To compare the partitioner's performance for different programs, timings for a hand coded block partitioned version in Table VII are also included. In the

TABLE III
PERFORMANCE OF SCHEDULE REUSE

(Time in Secs)	10K Mesh		53k Mesh		648 Atoms		
	Processors		Processors		Processors		
	8	16	32	64	4	8	16
No Schedule Reuse	161	94	301	189	707	384	227
Schedule Reuse	10.1	8.4	19.8	17.0	15.2	9.7	8.0

TABLE IV
UNSTRUCTURED MESH TEMPLATE—53K MESH—32 PROCESSORS

(Time in Secs)	Recursive Coordinate Bisection			Block Partition	
	Hand Coded	Compiler: No Schedule Reuse	Compiler Schedule Reuse	Hand Coded	Compiler Schedule Reuse
Partitioner	1.3	1.3	1.3	0.0	0.0
Inspector & Remap	3.3	286	3.4	3.2	3.4
Executor	13.9	13.9	15.1	36.6	38.2
Total	18.5	301	19.8	39.8	41.6

TABLE V
UNSTRUCTURED MESH TEMPLATE—53K MESH—32 PROCESSORS

(Time in Secs)	Inertial Bisection		Spectral Bisection	
	Hand Coded	Compiler Schedule Reuse	Hand Coded	Compiler: Schedule Reuse
Graph Generation	-	-	1.8	2.1
Partitioner	1.4	1.4	226	227
Inspector & Remap	3.1	3.3	3.1	3.2
Executor	14.7	16.1	12.5	13.4
Total	19.2	20.8	243	246

TABLE VI
PERFORMANCE OF COMPILER-LINKED COORDINATE BISECTION PARTITIONER WITH SCHEDULE REUSE

Tasks (Time in Secs)	10K Mesh		53k Mesh		648 Atoms		
	Processors		Processors		Processors		
	8	16	32	64	4	8	16
Partitioner	0.4	0.4	1.3	3.1	0.1	0.1	0.1
Inspector	0.4	0.3	0.9	0.5	2.2	1.2	0.7
Remap	1.4	0.9	2.5	1.7	4.8	2.6	1.5
Executor	7.9	6.9	15.1	11.7	8.1	5.8	5.7
Total	10.1	8.5	19.8	17.0	15.2	9.7	8.0

TABLE VII
COMPILER PERFORMANCE FOR BLOCK DISTRIBUTION WITH SCHEDULE REUSE

Tasks (Time in Secs)	10K Mesh		53k Mesh		648 Atoms		
	Processors		Processors		Processors		
	8	16	32	64	4	8	16
Inspector	0.5	0.3	1.2	0.7	2.7	1.5	0.8
Remap	1.2	0.8	2.2	1.0	4.5	2.6	1.5
Executor	17.6	12.6	38.2	28.6	10.3	7.6	7.3
Total	19.3	13.7	41.6	30.3	17.5	11.7	9.6

blocked version, each contiguous block of array elements are assigned to processors using the BLOCK distribution. The use of either a coordinate bisection partitioner or a spectral bisection partitioner led to a reduction factor of two to three in the executor time compared to the use of block partitioning. This example also points out the importance of the number of executor iterations and choice of partitioner. When compared to the RCB partitioner, the RSB partitioner is associated with faster time per executor iteration but also a significantly higher partitioning overhead. Irregular distribution of arrays performs significantly better than the existing BLOCK distribution supported by HPF.

C. Performance of Adaptive Problems

Table VIII presents experimental results for an application of the type described in Section II.A.2. Recall that this type of application alternates between two distinct computational phases. The first phase (*convection*) consists of structured calculations on a Cartesian mesh. The second phase (*reaction*) involves a set of local computations at each mesh point. The computational cost associated with the reaction phase varies between mesh points. Fig. 5 in Section II.A.2 depicts the computational structure of this type of application.

The presented results are for a simplified version of the Reactive Euler solver developed by James Weber at the University of Maryland. This algorithm computes the reaction rates of various gases, integrates the governing rate equations, and determines the new *number densities* in a hypersonic medium. The thermodynamic quantities, such as temperature, pressure, and specific heat ratio are evaluated as the reaction mechanism proceeds. The first phase of the Reactive Euler solver is an explicit Navier Stokes solver, while the second phase is an adaptive ordinary differential equation solver.

Fig. 13 depicts the load balancing strategy. In this simplified example, the mesh is represented as a one dimensional array. The array is partitioned into equal-size blocks (i.e., a BLOCK mapping). In order to ensure a good load balance during the reaction phase, only expensive reaction calculations are redistributed. Reaction calculations are redistributed based on the costs incurred in the previous time step. After the reaction phase, the remapped data are returned to their original positions.

TABLE VIII
PERFORMANCE OF COMBUSTION CODE WITH COMPILER-LINKED LOAD BASED PARTITIONER

Grid Size (Time in Sec.)	Processors	Hand Coded			Compiler Generated			No Load Balance
		Load Balance	Comp	Total	Load Balance	Comp	Total	Total
1024x32	16	4.1	19.4	23.5	4.3	19.4	23.7	117
	32	2.8	12.4	15.2	2.9	12.8	15.8	61
1024x128	32	8.5	34.1	42.9	8.5	34.6	43.4	397
	64	5.2	26.4	31.6	5.2	26.9	32.1	342

TABLE IX
PERFORMANCE OF COMPILER-LINKED PARTITIONERS

Total Time (in Secs)	10K Mesh		53k Mesh		648 Atoms			1024x32 Grid	
	Processors		Processors		Processors			Processors	
	8	16	32	64	4	8	16	16	32
Hand Coded	8.8	7.0	18.5	14.9	14.3	8.5	7.0	23.5	15.2
Compiler	10.1	8.5	19.8	17.0	15.2	9.7	8.0	23.7	15.8

Table VIII presents the performance of the second reaction phase for 100 cycles, and a comparison between hand coded and compiler generated codes. The *Load Balance* columns give the time taken to carry out the partitioner and remap the data. A bin-packing heuristic is used to balance the load in the combustion phase. The performance of the compiler generated code is almost as good as that of the hand coded version. Also note the performance improvements obtained when using a load based partitioner and adaptivity compared to performing no load balancing.

Finally, Table IX summarizes the compiler performance for all the codes and presents a comparison with the hand coded version. For all problems, the performance of the compiler generated code is within 15% of the hand coded version.

VII. RELATED WORK

Research has been carried out by von Hanxleden [16] on compiler-linked partitioners that decompose arrays based on distributed array element values; these are called *value-based decompositions*. The GEOMETRY construct can be viewed as a particular type of value based decomposition. Several researchers have developed programming environments that are targeted toward particular classes of irregular or adaptive problems. Williams [44] describes a programming environment (DIME) for calculations with unstructured triangular meshes using distributed memory machines. Baden [1] has developed a programming environment targeted towards particle computations. This programming environment provides facilities that support dynamic load balancing.

There are a variety of compiler projects targeting distributed memory multiprocessors: the Fortran D compiler projects at Rice and Syracuse [14], [4] and the Vienna Fortran compiler project [45] at the University of Vienna are two examples. The Jade project at Stanford [24], the DINO project at Colorado [37], Kathy Yelick's work [6] at Berkeley, and the CODE project at University of Texas, Austin, provide parallel programming environments. Runtime compilation methods have been employed in four compiler projects: the Fortran D project [20], the Kali project [23], Marina Chen's work at Yale [26], and the PARTI project [30], [38]. The Kali compiler was the first compiler to implement inspector/executor type runtime preprocessing [23], and the ARF compiler was the first compiler to support irregularly distributed arrays [38].

In earlier work, a strategy was outlined that would make it possible for compilers to generate compiler embedded connectivity based partitioners directly from marked loops [36]. The approach described here requires more input from the user and less compiler support. A short version of the techniques described in this paper appeared in a conference proceedings [35]. Support for irregular data distributions in HPF, using intrinsic functions, has been proposed by Ponnusamy et al. [34]. Recently, support for irregular data distribution has been implemented on the Vienna Fortran Compiler, using CHAOS runtime procedures, in collaboration with this research group.

VIII. Conclusions

This paper has described work that demonstrates two new mechanisms for dealing effectively with irregular computations. The first mechanism invokes a user specified mapping procedure using a set of compiler directives. The second mechanism is a simple conservative method that in many cases makes it possible for a compiler to recognize the potential for reusing previously computed results from inspectors (e.g., communication schedules, loop iteration partitions, and information that associates off-processor data copies with on-processor buffer locations).

The CHAOS procedures described here can be viewed as forming a portion of a portable, compiler independent, runtime support library. The CHAOS runtime support library contains procedures that

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

The prototype compiler has been tested on computational templates extracted from an unstructured mesh computational fluid dynamics code, a molecular dynamics code, and an hypersonic combustion code. The hand parallelized codes, where runtime support routines are embedded by hand, have been compared against the compiler generated codes. The compiler's performance on these templates was within 15% of the hand compiled codes.

In the current implementation, iteration partitioning of a Forall loop has been performed using the almost owner computes rule. In general, for data partitioning, a user or compiler should be able to specify a partitioner to perform iteration partitioning. Currently, primitives are been developed to couple iteration partitioners with Fortran 90 Forall loops.

The CHAOS procedures described in this paper are available for public distribution and can be obtained from netlib or from the anonymous ftp site hyena.cs.umd.edu.

Acknowledgments

The authors would like to thank Alan Sussman and Raja Das for many fruitful discussions; Shamik Sharma and Mustafa Uysal for their part in constructing the CHAOS runtime support; and Donna Meisel for careful proof reading of this manuscript.

The authors would like to thank Chuck Koelbel and Sanjay Ranka for many enlightening discussions about universally applicable partitioners and how to embed such partitioners in compilers. The authors would also like to thank Ken Kennedy and Seema Hiranandani for many useful discussions about integrating Fortran D runtime support for irregular problems. Special thanks go to Reinhard von Hanxleden for his helpful suggestions.

The authors would also like to gratefully acknowledge Zeki Bozkus and Tom Haupt for the time they spent explaining the internals of the Fortran 90D compiler. The authors would also like to thank Horst Simon for the use of his unstructured mesh partitioning software.

This work was sponsored in part by ARPA (NAG-1-1485), NSF (ASC 9213821) and ONR (N00014-93-1-0158, N00014-94-1-0907). Also supported by NASA Contract No. NAS1-19480 while author Saltz was in residence at ICASE, NASA Langley Research Center, Hampton, Virginia. Author Choudhary was also supported by NSF Young Investigator award (CCR-9357840) and matching funds Intel SSD division and IBM Corporation. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

References

[1] S. Baden, "Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors," *SIAM J. Sci. and Stat. Computation*, vol. 12, no. 1, Jan. 1991.

[2] S.T. Barnard and H. Simon, "A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," Technical Report RNR-92-033, NAS Systems Division, NASA Ames Research Center, Nov. 1992.

[3] M.J. Berger and S.H. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Trans. on Computers*, vol. 36, no. 5, pp. 570–580, May 1987.

[4] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, S. Ranka, and M.-Y. Wu. "Compiling Fortran 90D/HPF for distributed memory MIMD computers," *J. of Parallel and Distributed Computing*, vol. 21, no. 1, pp. 15–26, Apr. 1994.

[5] 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," *J. Computational Chemistry*, vol. 4, p. 187, 1983.

[6] S. Chakrabarti and K. Yelick, "Implementing an irregular application on a distributed memory multiprocessor," *Proc. of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, ACM SIGPLAN Notices, vol. 28, no. 7, May 1993.

[7] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," *Scientific Programming*, vol. 1, no. 1, pp. 31–50, Fall 1992.

[8] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," Technical Report 92-9, ICASE, NASA Langley Research Center, Mar. 1992.

[9] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and J. Saltz. "Software support for irregular and loosely synchronous problems," *Computing Systems in Eng.*, vol. 3, nos. 1-4, pp.

43–52, 1992. Papers presented at the Symp. on High-Performance Computing for Flight Vehicles, Dec. 1992.

[10] T.W. Clark, R. von Hanxleden, J.A. McCammon, and L.R. Scott, "Parallelization strategies for a molecular dynamics program," *Intel Supercomputer Univ. Partners Conf.*, Mt. Hood, Oreg., Apr. 1992.

[11] R. Das, D.J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy, "The design and implementation of a parallel unstructured Euler solver using software primitives," *AIAA J.*, vol. 32, no. 3, pp. 489–496, Mar. 1994.

[12] R. Das, J. Saltz, and R. von Hanxleden, "Slicing analysis and indirect access to distributed arrays," *Proc. of the Sixth Workshop on Languages and Compilers for Parallel Computing*, pp. 152–168. New York: Springer-Verlag, 1993. Also available as Univ. of Maryland Technical Report CS-TR-3076 and UMIACS-TR-93-42.

[13] D. Loveman, ed., "High performance Fortran language specification, version 1.0," Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice Univ., Jan. 1993.

[14] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D language specification," Dept. of Computer Science, COMP TR90-141, Rice Univ., Dec. 1990.

[15] S. Hammond and T. Barth, "An optimal massively parallel Euler solver for unstructured grids," *AIAA J., AIAA Paper 91-0441*, Jan. 1991.

[16] R. von 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 Univ., Dec. 1993.

[17] R. von Hanxleden and L.R. Scott, "Load balancing on message passing architectures," *J. of Parallel and Distributed Computing*, vol. 13, pp. 312–324, 1991.

[18] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," Technical Report SAND 92-1460, Sandia National Laboratory, Albuquerque, N. Mex., Sept. 1992.

[19] High Performance Fortran Forum, "High performance Fortran language specification," *Scientific Programming*, vol. 2, nos. 1-2, pp. 1–170, 1993.

[20] S. Hiranandani, K. Kennedy, and C. Tseng, "Compiler support for machine-independent parallel programming in Fortran D," J. Saltz and P. Mehrotra, eds., *Compilers and Runtime Software for Scalable Multiprocessors*. Amsterdam, The Netherlands: Elsevier, 1991.

[21] A. Jameson, T.J. Baker, and N.P. Weatherhill, "Calculation of inviscid transonic flow over a complete aircraft," *AIAA paper 86-0103*, Jan. 1986.

[22] B.W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical J.*, vol. 49, no. 2, pp. 291–307, Feb. 1970.

[23] C. Koelbel, P. Mehrotra, and J. Van Rosendale, "Supporting shared data structures on distributed memory architectures," *Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 177–186, Mar. 1990.

[24] M. Lam, E.E. Rothberg, and M.E. Wolf, "The cache performance and optimizations of block algorithms," *Proc. of the Fourth Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pp. 63–74. Washington, D.C.: ACM Press, 1991.

[25] W.E. Leland, "Load-balancing heuristics and process behavior," *Proc. of Performance 86 and ACM SIGMETRICS 86*, pp. 54–69, 1986.

[26] L.C. Lu and M.C. Chen, "Parallelizing loops with indirect array references or pointers," *Proc. of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, Calif., Aug. 1991.

[27] N. Mansour, "Physical optimization algorithms for mapping data to distributed-memory multiprocessors," Technical Report, PhD dissertation, School of Computer Science, Syracuse Univ., 1992.

[28] D.J. Mavriplis, "Adaptive mesh generation for viscous flows using delaunay triangulation," *J. of Computational Physics*, vol. 90, no. 2, pp. 271–291, 1990.

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

[30] R. Mirchandaney, J.H. Saltz, R.M. Smith, D.M. Nicol, and K. Crowley, "Principles of runtime support for parallel processors," *Proc. of the 1988 ACM Int'l Conf. on Supercomputing*, pp. 140–152, July 1988.

[31] B. Nour-Omid, A. Raefsky, and G. Lyzenga, "Solving finite element equations on concurrent computers," *Proc. of Symp. on Parallel Computations and the Impact on Mechanics*, Boston, Dec. 1987.

[32] G. Patnaik, K.J. Laskey, K. Kailasanath, E.S. Oran, and T.V. Brun, "FLIC—a detailed, two-dimensional flame model," NRL Report 6555, Naval Research Laboratory, Washington, D.C., Sept. 1989.

[33] R. Ponnusamy, "A manual for the CHAOS runtime library," Technical Report TR93-105, Computer Science Dept., Univ. of Maryland, Dec. 1993 (available at anonymous ftp site hpsl.cs.umd.edu).

[34] R. Ponnusamy, Y.-S. Hwang, R. Das, J. Saltz, A. Choudhary, and G. Fox, "Supporting irregular distributions using data-parallel languages," *IEEE Parallel and Distributed Technology*, vol. 3, no. 1, pp. 12-24, Spring 1995.

[35] R. Ponnusamy, J. Saltz, and A. Choudhary, "Runtime-compilation techniques for data partitioning and communication schedule reuse," *Proc. Supercomputing '93*, pp. 361–370. Los Alamitos, Calif.: IEEE CS Press, Nov. 1993. Also available as Univ. of Maryland Technical Report CS-TR-3055 and UMIACS-TR-93-32.

[36] R. Ponnusamy, J. Saltz, C. Koelbel, and A. Choudhary, "A runtime mapping scheme for irregular problebms," *Proc. of the Scalable High Performance Computing Conf. (SHPCC-92)*, pp. 216–219, Williamsburg, Va., Apr. 1992. Los Alamitos, Calif.: IEEE CS Press, 1992.

[37] M. Rosing, R.B. Schnabel, and R.P. Weaver, "The DINO parallel programming language," *J. of Parallel and Distributed Computing*, vol. 13, no. 1, pp. 30–42, Sept. 1991.

[38] J. Saltz, H. Berryman, and J. Wu, "Runtime compilation for multiprocessors," *Concurrency: Practice and Experience*, vol. 3, no. 6, pp. 573–592, 1991.

[39] H. Simon, "Partitioning of unstructured mesh problems for parallel processing," *Proc. of the Conf. on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Pergamon Press, 1991.

[40] W.F. van Gunsteren and H.J.C. Berendsen, "Gromos: Groningen molecular simulation software," Technical Report, Laboratory of Physical Chemistry, Univ. of Groningen, Nijenborgh, The Netherlands, 1988.

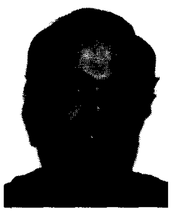
[41] P. Venkatkrishnan, J. Saltz, and D. Mavriplis, "Parallel preconditioned iterative methods for the compressible navier stokes equations," *12th Int'l Conf. on Numerical Methods in Fluid Dynamics*, Oxford, England, July 1990.

[42] N.P. Weatherill, "The generation of unstructured grids using dirichlet tessalations," Report MAE 1715, Princeton Univ., July 1985.

[43] P.K. Weiner and P.A. Kollman, "Amber:assisted model building with energy refinement: A general program for modeling molecules and their interactions," *J. of Computational Chemistry*, vol. 2, p. 287, 1981.

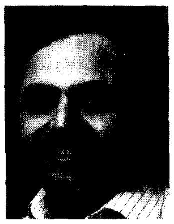
[44] R. Williams, "Performance of dynamic load balancing algorithms for unstructured mesh calculations," *Concurrency, Practice and Experience*, vol. 3, no. 5, pp. 457–482, Feb. 1991.

[45] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald, "Vienna Fortran—a language specification," Report ACPC-TR92-4, Austrian Center for Parallel Computation, Univ. of Vienna, Vienna, Austria, 1992.



Ravi Ponnusamy received a Bachelor of Engineering degree in computer science and engineering from Anna University, Madras, India, in 1987 and a PhD in computer science from Syracuse University in 1994. He is a research associate in the Computer Engineering Department at Syracuse University. Prior to this, he was a faculty research assistant in the Computer Science Department at the University of Maryland, College Park. He has been designing and developing toolkits and techniques for high performance Fortran compilers to produce efficient parallel code for large-scale scientific applications. A paper he coauthored received the Best Student Paper award at the Supercomputing 1992 conference. His research interests include parallel I/O, parallelizing compilers, supercomputer applications, and performance evaluation.

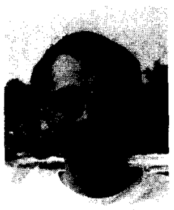
Joel Saltz is associate professor of computer science and director of the High-Performance Systems Software Laboratory, University of Maryland. He leads a research group at the University of Maryland, College Park, whose goal is to develop methods that will make it possible to produce portable compilers that generate efficient multiprocessor code for irregular scientific problems, i.e., problems that are unstructured, sparse, adaptive, or block structured. He collaborates with a wide variety of applications researchers from areas such as computational fluid dynamics, computational chemistry, computational biology, environmental sciences, structural mechanics, and electrical power grid calculations. He came to the University of Maryland after spending three years at ICASE at the NASA-Langley Research Center as lead computer scientist and three years at Yale University as an assistant professor.



Alok Choudhary received his BE (Hons.) in electrical and electronics engineering from Birla Institute of Technology and Science, Pilani, India, in 1982, his MS from the University of Massachusetts, Amherst, in 1986, and his PhD from the University of Illinois, Urbana-Champaign, in electrical and computer engineering in 1989. He joined the Department of Electrical and Computer Engineering at Syracuse University in 1989, where he is currently an associate professor. He was a visiting scientist at IBM's T.J. Watson Research Center during the summers of 1987 and 1988. He worked as a system analyst and designer from 1982 to 1984.

Choudhary's main research interests are in parallel and distributed processing and in software development environments for parallel computers, including compilers and runtime support, parallel computer architectures, and parallel I/O systems. He has published over 60 journal and conference papers in the above areas. He has also coauthored *Parallel Architectures and Parallel Algorithms for Integrated Vision Systems*, published by Kluwer Academic Publishing Co., 1990.

Choudhary was a program cochair for the International Conference on Parallel Processing, 1993. He served as a guest editor for IEEE *Computer* and the *Journal of Parallel and Distributed Computing* (JPDC). He is currently a subject-area editor of JPDC. He is a member of the IEEE Computer Society and the Association for Computing Machinery. He received an IEEE Engineering Foundation Award in 1990 and the NSF Young Investigator Award in 1993.



Yuan-Shin Hwang received his BS and MS in electrical engineering from the National Tsing Hua University, Hsinchu, Taiwan, in 1987 and 1989, respectively. He is pursuing his PhD degree in computer science at the University of Maryland, College Park, where he is currently 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.

Geoffrey Fox earned his PhD in theoretical physics from Cambridge University in 1967. He is currently professor of computer science and physics at Syracuse University and director of the Northeast Parallel Architectures Center. He is an internationally recognized expert in the use of parallel architectures and the development of concurrent algorithms. He leads a major project to develop prototype high-performance Fortran (Fortran 90D) compilers. He is also a leading proponent for the development of computational science as an academic discipline and a scientific method. His research on parallel computing has focused on development and use of this technology to solve large-scale computational problems. He directs InfoMall, which is focused on accelerating the introduction of parallel computing into New York State industry. A current focus is large-scale multimedia information systems accessed by the national digital highway. He coauthored *Solving Problems on Concurrent Processors* and edits *Concurrency: Practice and Experience* and the *International Journal of Modern Physics: C*. His research experience includes work at the Institute for Advanced Study at Princeton; Lawrence Berkeley Laboratory; Cavendish Laboratory at Cambridge; Brookhaven National Laboratory; and Argonne National Laboratory. He has served as dean for educational computing and assistant provost for computing at Caltech.