Runtime Compilation Techniques for Data Partitioning and Communication Schedule Reuse*

Ravi Ponnusamy[†‡] Joel Saltz[†] Alok Choudhary[‡]

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

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

Abstract

In this paper, we describe two new ideas by which HPF compiler can deal with irregular computations effectively. The first mechanism invokes a user specified mapping procedure via a set of compiler directives. The directives allow the user to use program arrays to describe graph connectivity, spatial location of array elements and computational load. The second is a simple conservative method that in many cases enables a compiler to recognize that it is possible to reuse previously computed results from inspectors (e.g. communication schedules, loop iteration partitions, information that associates off-processor data copies with on-processor buffer locations). We present performance results for these mechanisms from a Fortran 90D compiler implementation.

1 Introduction

In sparse and unstructured problems the data access pattern is determined by variable values known only at runtime. In these cases, programmers carry out preprocessig to partition work, map data structures and schedule the movement of data between the memories of processors. The code needed to carry out runtime preprocessing can also be generated by a distributed memory compiler in a process we call *runtime compilation* [23]. In this paper, we present methods and a prototype implementation where we demonstrate techniques that make it possible for compilers to efficiently handle irregular problems coded using a set of language extensions closely related to Fortran D [10] or Vienna Fortran [28].

On distributed memory architectures, loops with indirect array accesses can be handled by transforming the original loop into two sequences of code: an *inspector* and an *executor*. The inspector partitions loop iterations, allocates local memory for each unique off-processor distributed array element accessed by a loop, and builds a communication schedule to prefetch required off-processor data. In the executor phase, the actual communication and computation are carried out [21]. The ARF compiler [26] and KALI compiler [16] used this kind of transformation to handle loops with indirectly referenced arrays (*irregular loops*).

We propose a simple conservative method that often makes it possible to reuse previously computed results from inspectors (e.g. communication schedules, loop iteration partitions, information that associates off-processor data copies with on-processor buffer locations). The compiler generates code that, at runtime, maintains a record of when a Fortran 90D loop or intrinsic may have written to a distributed array that is used to indirectly reference another distributed array. In this scheme, each inspector checks this runtime record to see whether any indirection arrays may have been modified since the last time the inspector was invoked.

In distributed memory machines, large data arrays need to be partitioned between local memories of processors. 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 distributed machine. It is frequently advantageous to partition distributed arrays in an irregular manner. For instance, the way in which the nodes of an irregular computational mesh are numbered frequently does not have a useful correspondence to the connectivity pattern of the mesh. When we partition the data structures in such a problem in a way that minimizes interprocessor communication, we may need to assign arbitrary array elements to each processor. In recent years promising heuristics have been developed and tradeoffs associated with the different partitioning methods have been studied [24, 25, 19, 17, 2, 13].

We have implemented the runtime support and compiler transformations needed to allow users to specify the information needed to produce a customized distribution function. In our view, this information can consist of a description of graph connectivity, spatial location of array elements and information that associates array elements with computational load. Based on user directives the compiler produces code that, at runtime, generates a standardized representation of the above information, and then passes this standardized representation to a (user specified) partitioner. The compiler also generates code that, at runtime, produces a data structure that is used to partition loop it-

*This work was sponsored in part by ARPA (NAG-1-1485), NSF (ASC 9213821) and ONR (SC292-1-22913). Author Choudhary was also supported by NSF Young Investigator award (CCR-9357840). The content of the information does not necessarily reflect the position of the policy of the Government and no official endorsement should be inferred

```
C Single statement loop L1
   FORALL i = 1, N
   y(ia(i)) = x(ib(i)) + ... x(ic(i))
   END FORALL
C Sweep over edges: Loop L2
   FORALL i = 1,N
   REDUCE (ADD, y(end_pt1(i)),
   f(x(end_pt1(i)), x(end_pt2(i))))
   REDUCE (ADD, y(end_pt2(i)),
   g(x(end_pt1(i)), x(end_pt2(i))))
   END FORALL
```

Figure 1: Example Irregular Loops

erations. To our knowledge, the implementation described in this paper is the first distributed memory Fortran compiler to provide this kind of support. We also note that in the Vienna Fortran [28] language definition, a user can also specify a customized distribution function. The runtime support and compiler transformation strategies described here can also be applied to Vienna Fortran.

We will describe the runtime support, compiler transformations and language extensions required to provide the new capabilities described above. We assume that irregular accesses are carried out in the context of a single or multiple statement loop where the only loop carried dependencies allowed are left hand side reductions (e.g. addition, accumulation, max, min, etc). We also assume that irregular array accesses occur as a result of a single level of indirection with a distributed array that is indexed directly by the loop index.

In the example loops shown in Figure 1, we employ Fortran D syntax to depict two loops. The first loop is a single statement loop with indirect array references without dependencies. The second loop is a loop in which we carry out reduction operations. The second loop is similar to those loops found in unstructured computational fluid dynamics codes and molecular dynamics codes. We use this loop to demonstrate our runtime procedures and compiler transformations in the following sections.

We have implemented our methods as part of the Fortran 90D compiler being developed by Syracuse University [9]. Our implementation results on simple templates reveal that the performance of the compiler generated code is within 10% of the hand parallelized version.

This paper is organized as follows. We set the context of the work in Section 2. In Section 3, we describe the runtime technique to save communication schedules. In Section 4 we describe the procedures used to couple data and loop iteration partitioners to compilers. In Section 5 we present an overview of our compiler effort. We describe the transformations which generate the standard data structure and describe the language extensions we use to control compiler-linked runtime partitioning. In Section 6 we present performance data to characterize the performance of our methods. We briefly discuss related work in Section 7 and we conclude in Section 8.

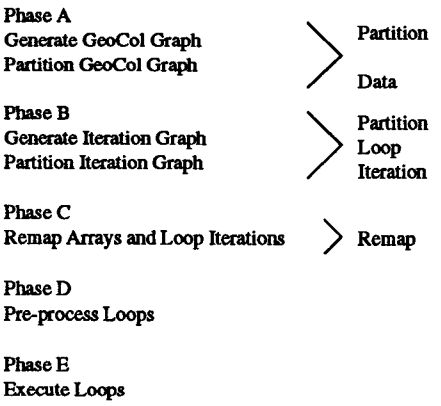

Figure 2: Solving Irregular Problems

2 Overview

2.1 Overview of CHAOS

We have developed efficient runtime support 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. The CHAOS library is a superset of the earlier PARTI library [21, 26, 23].

Solving concurrent irregular problems on distributed memory machines using our runtime support, involves five major steps (Figure 2). The first three steps in the figure concern mapping data and computations onto processors. We provide a brief description of these steps here, and will discuss them in detail in later sections.

Initially, the distributed arrays are decomposed in a known regular manner. In Phase A of Figure 2, CHAOS procedures can be called to construct a graph data structure (the GeoCoL data structure) using the data access patterns associated with a particular set of loops. The GeoCoL graph data structure is passed to a partitioner. The partitioner calculates how data arrays should be distributed.

In Phase B, the newly calculated array distributions are used to decide how loop iterations are to be partitioned among processors. This calculation takes into account loop data access patterns. In Phase C we carry out the actual remapping of arrays and loop iterations.

In Phase D, we carry out the preprocessing needed to (1) coordinate interprocessor data movement, (2) manage the storage of, and access to, copies of off-processor data, and (3) support a shared name space. This preprocessing involves generating communication *schedules*, translating

array indices to access local copies of off-processor data and allocating local buffer space for copies of off-processor data. It is also necessary to retrieve globally indexed but irregularly distributed data-sets from the numerous local processor memories. Finally, in Phase E we use information from the earlier phases to carry out the necessary computation.

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 [23] aimed at distributed memory multiprocessors.

2.2 Overview of Existing Language Support

The data decomposition directives we employ for irregular problems will be presented in the context of Fortran D. While our work will be 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 and HPF. Vienna Fortran, Fortran D and HPF (evolved from Fortran D and Fortran 90) provide a rich set of data decomposition specifications; a definition of such language extensions may be found in [10, 8]. These languages, as currently specified, require that users explicitly define how data is to be distributed.

Fortran D can be used to *explicitly* specify an irregular inter-processor partition of distributed array elements. In Figure 3, we present an example of such a Fortran D declaration. In Fortran D, one declares a template called a *distribution* which 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 processors.

Fortran D provides the user with a choice of several regular distributions. In addition, a user can explicitly specify how a distribution is to be mapped onto processors. A specific array is associated with a distribution using the Fortran D statement **ALIGN**. In statement S3, of Figure 3, two of size `N` each, one dimensional decompositions 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.

```
....
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
....
```

Figure 3: Fortran D Irregular Distribution

The difficulty with the declarations depicted in Figure 3 is that *it is not obvious how to partition the irregularly distributed array.* The *map* array which gives the distribution pattern of `irreg` has to be generated separately by running a partitioner. 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 no standard interface between the partitioners and the application codes.

3 Communication Schedule Reuse

The cost of carrying out an inspector (phases B, C and D in Figure 2) can be amortized when the information produced by the inspector is computed once and then used repeatedly. Compile time analysis needed to reuse inspector communication schedules is touched upon in [12, 7].

We propose a simple conservative method that in many cases allows us to reuse the results from inspectors. The results from an inspector for loop L can be reused as long as:

- distributions of data arrays referenced in loop L have remained unchanged since the last time the inspector was invoked, and
- there is no possibility that indirection arrays associated with loop L have been modified since the last inspector invocation.

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

In this presentation, we assume that we are carrying out an inspector for a forall loop. We also assume 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.

A data access descriptor (DAD) for a distributed array contains (among other things) the current distribution type of the array (e.g. block, cyclic, irregular) 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 our scheme, we will maintain a *global* data structure that contains information on when *any array* with a given DAD may have been modified.

We maintain a global variable **n_mod** which represents the cumulative number of Fortran 90D loops, array intrinsics or statements that have modified any distributed array. Note that we are not counting the number of assignments to the distributed array, instead we are counting the number of times the program will execute any block of code that writes to a distributed array. **n_mod** may be viewed as a global time stamp. Each time we modify an array **a** with a given data access descriptor DAD(**a**), we update a global data structure **last_mod** to associate DAD(**a**) with the current value of the global variable **n_mod** (i.e. the current global timestamp). Thus when a loop, array intrinsic or statement modifies **a** we set **last_mod(DAD(a)) = n_mod**. If the array **a** is remapped, it means that DAD(**a**) changes. In this case, we increment **n_mod** and then set **last_mod(DAD(a)) = n_mod**.

The first time an inspector for a forall loop L is carried out, it must perform all the preprocessing. Assume that L has m data arrays x_L^i, $1 \leq i \leq m$, and n indirection arrays, ind_L^j, $1 \leq j \leq n$. Each time an inspector for L is carried out, we store the following information:

DAD(x_L^i) for each unique data array x_L^i , for $1 \leq i \leq m$, and

DAD(ind_L^j) for each unique indirection array ind_L^j, for $1 \leq j \leq n$ and

last_mod(DAD(ind_L^j)), for $1 \leq j \leq n$.

We designate the values of DAD(x_L^i), DAD(ind_L^j) and last_mod(DAD(ind_L^j)) stored by L's inspector as L.DAD(x_L^i), L.DAD(ind_L^j) and L.last_mod(DAD(ind_L^j)).

For a given data array x_L^i and indirection array ind_L^j in a forall loop L, we maintain *two sets of data access descriptors*. For instance, we maintain,

- DAD(x_L^i) the current global data access descriptor associated with x_L^i, and
- L.DAD(x_L^i) is a record of the data access descriptor that was associated with x_L^i when L carried out its last inspector.

For each indirection array ind_L^j, we also maintain two timestamps:

- last_mod(DAD(ind_L^j) is the global timestamp associated with the current data access descriptor of ind_L^j and,
- L.last_mod(DAD(ind_L^j)) is the global timestamp of data access descriptor DAD(ind_L^j), last recorded by L's inspector.

After the first time L's inspector has been executed, the following checks are performed before the subsequent executions of L. If any of the following conditions is false, the inspector must be repeated for L.

1. DAD(x_L^i) == L.DAD(x_L^i), $1 \leq i \leq m$
2. DAD(ind_L^j) == L.DAD(ind_L^j), $1 \leq j \leq n$
3. last_mod(DAD(ind_L^j)) ==

   L.last_mod(L.DAD(ind_L^j)), $1 \leq j \leq n$.

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 6). Calculations in such codes primarily occur in loops or Fortran 90 array intrinsics, so we need to record modifications to a DAD *once* per loop or array intrinsic call.

We employ the same method to track possible changes to arrays used in the construction of the data structure produced at runtime to link partitioners with programs. We call this data structure a GeoCoL graph, and it will be described in Section 4.1.1. This approach makes it simple for our compiler to avoid generating a new GeoCoL graph and carrying out a potentially expensive repartition when no change has occurred.

We could further optimize our inspector reuse mechanism by noting that there is no need to record modifications to *all* distributed arrays. Instead, we could limit ourselves to recording possible modifications of the sets of arrays that have the same data access descriptor as an indirection array. Such optimization will require interprocedural analysis to identify the sets of arrays that must be tracked at runtime. Future work will include exploration of this optimization.

4 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 [3]. Consequently, we partition *both* distributed arrays and loop iterations using a two-phase approach (Figure 2). In the first phase, termed a "data partitioning" phase, distributed arrays are partitioned. In the second phase, called "workload 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 two phases.

4.1 Data Partitioning

When we partition distributed arrays, we have not yet assigned loop iterations to processors. We assume that we will partition loop iterations so as to attempt to minimize non-local distributed array references. Our approach to data partitioning 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 - we call this the *almost owner computes rule.*

There are many partitioning heuristics methods available based on physical phenomena and physical proximity [24, 2, 25, 13]. Currently these partitioners must be coupled to user programs in a manual fashion. This manual coupling is particularly troublesome and tedious when we 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.

4.1.1 Interface Data Structures for Partitioners

We link partitioners to programs by using a data structure that stores information on which data partitioning is to be based. Data structure partitioners can make use of different kinds of program information. Some partitioners operate on data structures that represent undirected graphs [24],[15], [19]. Graph vertices represent array indices, graph edges represent dependencies. Consider the example loops in Figure 1. In both loops, the graph vertices represent the N elements of arrays **x** and **y**. The graph edges in the first loop of Figure 1 are the union of:

edges linking vertices ia(i) and ib(i), i = 1,N

edges linking vertices ia(i) and ic(i), i = 1,N

The graph edges in the second loop of Figure 1 are the union of edges linking vertices end_pt1(i) and end_pt2(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. We can assign each graph vertex a set of coordinates that describe its spatial location. These spatial locations can be used to partition data structures [2, 22].

Vertices may also be assigned weights to represent estimated computational costs. In order to accurately estimate computational costs, we need information on how work will be partitioned. One way of deriving weights is to make the implicit assumption that an owner compute 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. This results in a graph with unit weights in the first loop in Figure 1 . The weight associated with a vertex in the second loop of Figure 1 would be proportional to the degree of the vertex when functions f and g have identical computational costs. Vertex weights can be used as a sole partitioning criterion in "embarrassingly parallel problems", problems in which computational costs dominate.

A given partitioner can make use of combinations of connectivity, geometrical or weight information. For instance, we find that it is sometimes 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 [5].

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

4.1.2 Generating GeoCoL Data Structure

We propose a directive **CONSTRUCT** that can be employed to direct a compiler to generate a GeoCoL data structure. 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 xcord, ycord, and zcord. The GEOMETRY construct is closely related to the geometrical partitioning or *value based decomposition* directives proposed by von Hanxleden [11].

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

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

Here, a GeoCoL construct 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. Integer arrays **edge_list1** and **edge_list2** list the vertices associated with each of E graph edges.

C$ **CONSTRUCT** G3 (N, **LINK**(E, edge_list1, edge_list2))

The keyword **LINK** is used to specify the edges associated with the GeoCoL graph.

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

C$ **CONSTRUCT** G4 (N, **GEOMETRY**(3, xcord, ycord, zcord), **LINK**(E, edge_list1, edge_list2))

```
      REAL*8 x(nnode),y(nnode)
      INTEGER end_pt1(nedge), end_pt2(nedge)
S1    DYNAMIC, DECOMPOSITION reg(nnode), reg2(nedge)
S2    DISTRIBUTE reg(BLOCK), reg2(BLOCK)
S3    ALIGN x,y with reg
S4    ALIGN end_pt1, end_pt2 with reg2
      ....
      call read_data(end_pt1, end_pt2, ...)
S5    CONSTRUCT G (nnode, LINK(nedge,end_pt1, end_pt2))
S6    SET distfmt BY PARTITIONING G USING RSB
S7    REDISTRIBUTE reg(distfmt)
C     Loop over edges involving x, y
      ....
C     Loop over faces involving x, y
```

Figure 4: Example of Implicit Mapping in Fortran 90D

Once the GeoCoL data structure is constructed, data partitioning is carried out. We assume there are **P** processors:

1. At compile time *dependency coupling code* is generated. This code generates calls to the runtime support that, when the program executes, generates the **GeoCoL** data structures,
2. The **GeoCoL** data structure is passed to a *data partitioning* procedure that partitions the **GeoCoL** into **P** subgraphs.
3. The **GeoCoL** vertices assigned to each subgraph specify an irregular distribution.

The GeoCoL data structure is constructed with the initial default distribution of distributed arrays. Once we have the new distribution given by the partitioner, we redistribute the arrays based on the new distribution. A communication schedule is built and used to redistribute the arrays from the default to the new distribution.

4.2 Linking Data Partitioners

In Figure 4 we illustrate a possible set of partitioner coupling directives for the loop L2 in Figure 1. We use statements S1 to S4 (Figure 4) to produce a default initial distribution of arrays x and y and the indirection array in loop L1, *end_pt*. The statements S5 and S6 directs 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 relations between distributed arrays x and y in loop L1 and the relationship is provided by using the keyword **LINK** in the

```
      ....
S'5   CONSTRUCT G (nnode, GEOMETRY(3, xc, yc, zc))
S'6   SET distfmt BY PARTITIONING G USING RCB
S'7   REDISTRIBUTE reg(distfmt)
C     Loop over edges involving x, y
      ....
C     Loop over faces involving x, y
```

Figure 5: Example of Implicit Mapping using Geometric Information in Fortran 90D

CONSTRUCT statement. The statement S6 in the figure calls the partitioner RSB (recursive spectral bisection) with GeoCoL as input. The user will be provided a library of commonly available partitioners and the user can choose any one one of them. Also, the user can link a customized partitioner as long as the calling sequence matches. Finally, the distributed arrays are remapped in statement S7 using the new distribution returned by the partitioner.

Figure 5 illustrates code similar to that shown in Figure 4 except that here 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 S'5 specifies that the GeoCoL data structure is to be constructed using geometric information. S'6 specifies that recursive binary coordinate bisection is used to partition the data.

4.3 Loop Iteration Partitioning

Once we have partitioned data, we must partition computational work. One convention is to compute a program assignment statement S in the processor associated with 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 we may need to generate communication within loops even in the absence of loop carried dependencies. For example, consider the following loop:

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

This loop has a loop independent dependence between S1 and S2 but no loop carried dependencies. Were we to assign work 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)) $\neq$ OWNER(ia(i)).

An alternate convention is to assign all work associated with a loop iteration to a given processor. We have developed data structures and procedures to support iteration partitioning.

Our current default is to employ a scheme that places a loop iteration on the processor that is the home of the largest number of the iteration's distributed array references.

5 Compiler Support

In the previous section we presented directives a programmer can use to implicitly specify how data and loop iterations are to be partitioned between processors. In this section we outline compiler transformations used to carry out this implicitly defined work and data mapping. The compiler transformations generate code which embeds the CHAOS mapper coupler procedures.

We use the example in Figure 4 to show how the compiler procedures are embedded in the code. A (simplified) version of the compiler transformation is shown in Figure 6. We start with BLOCK array distributions. Statements S5 to S7 in Figure 4 are used to generate a data distribution. When the **CONSTRUCT** statement is encountered, the compiler generates code with embedded CHAOS procedure calls, during program execution, the CHAOS procedures generate the GeoCoL data structure. The GeoCoL data structure is then passed to an user specified partitioner. When the **REDISTRIBUTE** statement is encountered, CHAOS data remapping procedure calls are generated to move arrays (x and y) aligned with the initial distribution (reg) to the new distribution (distfmt).

Loop iterations are partitioned at runtime using the method described in Section 4.3 whenever a loop accesses at least one irregularly distributed array.

6 Experimental Results

6.1 Timing Results for Schedule Reuse

In this section, we present performance data for the schedule saving technique proposed in Section 3 for the Fortran 90D compiler implementation. These timings involve a loop over edges of an 3-D unstructured Euler solver [20] for 10K and 53K mesh points and an electrostatic force calculation loop in a molecular dynamics code for 648 atom water simulation [4]; the functionality of these loops is equivalent to the loop L2 in Figure 1. Table 1 depicts the performance results of compiler generated code with and without the schedule reuse technique for

---

Start with block distribution of arrays x, y and end_pts

Read Mesh (end_pt1, end_pt2, ...)

C$ **CONSTRUCT** G (nnode, **LINK** (nedge, end_pt1, end_pt2))

K1 Call CHAOS procedures to generate GeoCoL data structure

C$ **SET** distfmt BY **PARTITIONING** G **USING RSB**

K2 Pass GeoCoL to RSB graph partitioner

K3 Obtain new distribution format from the partitioner(distfmt)

C$ **REDISTRIBUTE** reg (distfmt)

K4 Remap arrays (x and y) aligned with distribution reg to distribution *distfmt*

Figure 6: Compiler Transformations for Implicit Data Mapping

unstructured mesh and molecular dynamics loops, varying the number of processors of Intel iPSC/860 hypercube. The table presents the execution time of the loops for 100 iterations with distributed arrays decomposed irregularly using a recursive binary dissection partitioner. The results shown in the table emphasizes the importance of schedule reuse.

6.2 Timing Results using the Mapper Coupler

In this section, we present data that compares the the costs incurred by the compiler generated mapper coupler procedures with the cost a hand embedded mapper coupler. These timings involve a loop over edges of an 3-D unstructured Euler solver and the electrostatic force calculation loop in a molecular dynamics code. The compiler-linked mapping technique was incorporated in the Fortran 90D compiler being developed at Syracuse University. We present the performance of our runtime techniques on different number of processors on an Intel iPSC/860.

To map arrays we employed two different kinds of parallel partitioners 1) a geometry based partitioner (coordinate bisection [2]) and 2) a connectivity based partitioner (recursive spectral bisection [24]). The performance of the compiler embedded mapper version and hand parallelized version are shown in Table 2.

In Table 2, *Partitioner* depicts the time needed to partition the arrays using the partitioners, *Executor* depicts the time needed to carry out the actual computation and communication, and *inspector* depicts the time taken to build the schedule. In Table 2, *Partition* under Spectral Bisection depicts the time needed to partition the GeoCoL graph data structure using a parallelized version of Simon's eigenvalue partitioner [24]. We partitioned the GeoCoL

Table 1: Performance of Schedule Reuse

(Time in Secs)	10K Mesh Processors			53k Mesh Processors			648 Atoms Processors		
	4	8	16	16	32	64	4	8	16
No Schedule Reuse	400	214	123	668	398	239	707	384	227
Schedule Reuse	17.6	10.8	7.7	30.4	23.0	17.4	15.2	9.7	8.0

Table 2: Unstructured Mesh Template - 53 K Mesh - 32 Processors

(Time in Secs)	Binary Coordinate Bisection			Block Partition	Spectral Bisection	
	Hand Coded	Compiler: No Schedule Reuse	Compiler Schedule Reuse	Hand Coded	Hand Coded	Compiler: Schedule Reuse
Graph Generation	-	-	-	-	2.2	2.2
Partitioner	1.6	1.6	1.6	0.0	258	258
Inspector, remap	4.3	379	4.2	4.7	4.1	4.0
Executor	16.4	17.2	17.2	54.7	13.2	13.9
Total	22.4	398	23.0	59.4	277.5	277.9

graph into a number of subgraphs equal to the number of processors employed. It should be noted that any common parallelized partitioner could be used as a mapper. The *graph generation* time depicts the time required to generate GeoCoL graph. The *Executor* time shown in Table 2 gives the time needed to carry out the executor phase for 100 times. The results shown in the table demonstrate that the performance of the compiler generated code is within 10% of the hand coded version. In table 4, we have included timings for a hand coded block partitioned version of the code in order to quantify the performance effects that arose from the decision to partition the problem. In the blocked version, we assigned each processor contiguous blocks of array elements. We see that the use of either a coordinate bisection partitioner or a spectral bisection partitioner lead to a factor of two to three reduction in the executor time compared to the use of block partitioning. This example also points out the importance of the number of executor iterations on which partitioner should be chosen. When compared to the recursive coordinate bisection partitioner, the recursive spectral bisection partitioner is associated with a faster time per executor iteration but a significantly higher partitioning overhead.

A detailed performance of the compiler-linked coordinate bisection for the unstructured mesh loop and the molecular dynamics loop is shown in Table 3. In Table 4, we present timing results for naive partition of arrays - we assigned each processor contiguous blocks of arrays to processors using BLOCK distribution allowed in HPF. Irregular distribution of arrays performs much better than the existing BLOCK distribution supported by HPF.

7 Related Work

Research has been carried out by von Hanxleden [11] on compiler-linked partitioners which decompose arrays based on distributed array element values, these are called *value based decompositions*. Our GEOMETRY construct can be viewed as a particular type of value based decomposition. Several researchers have developed programming environments that are targeted towards particular classes of irregular or adaptive problems. Williams [25] 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 targeted at distributed memory multiprocessors [27, 16]. Jade project at Stanford, DINO project at Colorado and CODE project at Austin provide parallel programming environments. Runtime compilation methods are employed in four compiler projects; the Fortran D project [14], the Kali project [16], Marina Chen's work at Yale [18] and our PARTI project [21, 26, 23]. The Kali compiler was the first compiler to implement inspector/executor type runtime preprocessing [16] and the ARF compiler was the first compiler to support irregularly distributed arrays [26].

In earlier work, several of the authors of the current paper outlined a strategy (but did not attempt a compiler implementation) that would make it possible for compilers to generate compiler embedded connectivity based partitioners directly from marked loops [6]. The approach described here requires more input from the user and lesser compiler support.

8 Conclusions

In this paper, we have described and and presented timing data for a prototype Fortran 90D compiler implementation. The work described here demonstrates two new

Table 3: Performance of Compiler-linked Coordinate Bisection Partitioner with Schedule Reuse

Tasks (Time in Secs)	10K Mesh Processors			53k Mesh Processors			648 Atoms Processors		
	4	8	16	16	32	64	4	8	16
Partitioner	0.6	0.6	0.4	1.8	1.6	2.5	0.1	0.1	0.1
Inspector	1.2	0.6	0.4	2.0	1.2	0.7	2.2	1.2	0.7
Remap	3.1	1.6	0.9	5.1	3.0	1.9	4.8	2.6	1.5
Executor	12.7	7.0	6.0	21.5	17.2	12.3	8.1	5.8	5.7
Total	17.6	10.8	7.7	30.4	23.0	17.4	15.2	9.7	8.0

Table 4: Performance of Block Partitioning with Schedule Reuse

Tasks (Time in secs)	10K Mesh Processors			53k Mesh Processors			648 Atoms Processors		
	4	8	16	16	32	64	4	8	16
Inspector	1.5	0.9	0.5	3.9	1.9	1.0	2.7	1.5	0.8
Remap	3.1	1.6	0.8	4.9	2.8	1.7	4.5	2.6	1.5
Executor	26.0	20.8	14.7	74.1	54.7	35.3	10.3	7.6	7.3
Total	30.4	23.3	16.0	82.9	59.4	38.0	17.5	11.7	9.6

ideas for dealing effectively with irregular computations. The first mechanism invokes a user specified mapping procedure using a set of directives. The second 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, information that associates off-processor data copies with on-processor buffer locations).

We view the CHAOS procedures described here as forming a portion of a portable, compiler independent, runtime support library. The CHAOS runtime support library contains procedures that

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

We consider our work to be a part of the ARPA sponsored integrated effort towards developing powerful compiler independent runtime support for parallel programming languages. The runtime support can be employed in other High Performance Fortran type compilers, and in fact, a subset of the runtime support described here has been incorporated into the Vienna Fortran compiler.

We tested our prototype compiler on computational templates extracted from an unstructured mesh computational fluid dynamics code [20] and from a molecular dynamics code [4]. We embedded our runtime support by hand and compared its performance against the compiler generated code. The compiler's performance on these templates was within about 10% of the hand compiled code.

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 and for help in proofreading. The authors would like to thank Geoffrey Fox, Chuck Koelbel and Sanjay Ranka for many enlightening discussions about universally applicable partitioners and how to embed such partitioners into compilers; we would also like to thank Chuck Koelbel, Ken Kennedy and Seema Hiranandani for many useful discussions about integrating into Fortran-D runtime support for irregular problems. Our special thanks go to Reinhard von Hanxleden for his helpful suggestions.

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

References

[1] S. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *SIAM J. Sci. and Stat. Computation.*, 12(1), January 1991.

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

[3] Harry Berryman, Joel Saltz, and Jeffrey Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

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

[5] T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott. Parallelization strategies for a molecular dynamics program. In *Intel Supercomputer University Partners Conference*, Timberline Lodge, Mt. Hood, OR, April 1992.

[6] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems - data copy reuse and runtime partitioning. In *Compilers and Runtime Software for Scalable Multiprocessors, J. Saltz and P. Mehrotra Editors*, Amsterdam, The Netherlands, 1992. Elsevier.

[7] R. Das and J. H. Saltz. Program slicing techniques for compiling irregular problems. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland , OR, August 1993.

[8] D. Loveman (Ed.). Draft High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, January 1993.

[9] Z. Bozkus et al. Compiling fortran 90d/hpf for distributed memory mimd computers. Report SCCS-444, NPAC, Syracuse University, March 1993.

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

[11] R. v. Hanxleden. Compiler support for machine independent parallelization of irregulr problems. Technical report, Center for Research on Parallel Computation, 1992.

[12] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[13] R. v. Hanxleden and L. R. Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13:312–324, 1991.

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

[15] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[16] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186. ACM, March 1990.

[17] W. E. Leland. Load-balancing heuristics and process behavior. In *Proceedings of Performance 86 and ACM SIGMETRICS 86*, pages 54–69, 1986.

[18] L. C. Lu and M.C. Chen. Parallelizing loops with indirect array references or pointers. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[19] N. Mansour. Physical optimization algorithms for mapping data to distributed-memory multiprocessors. Technical report, Ph.D. Dissertation, School of Computer Science,Syracuse Universit y, 1992.

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

[21] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, July 1988.

[22] B. Nour-Omid, A. Raefsky, and G. Lyzenga. Solving finite element equations on concurrent computers. In *Proc. of Symposium on Parallel Computations and theis Impact on Mechanics*, Boston, December 1987.

[23] J. Saltz, H. Berryman, and J. Wu. Runtime compilation for multiprocessors. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.

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

[25] R. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency, Practice and Experience*, 3(5):457–482, February 1991.

[26] J. Wu, J. Saltz, S. Hiranandani, and H. Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume 2, pages 26–30, 1991.

[27] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

[28] 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, University of Vienna, Vienna, Austria, 1992.