10-1991

Distributed Memory Compiler Methods for Irregular Problems -- Data Copy Reuse and Runtime Partitioning

Raja Das
Syracuse University

Ravi Ponnusamy
NASA Langley Research Center, ICASE

Joel Saltz
Syracuse University

Dimitri Mavriplis
NASA Langley Research Center, ICASE

SU-CIS-91-36

Distributed Memory Compiler Methods for Irregular Problems -- Data Copy Reuse and Runtime Partitioning

R.Das, R.Ponnusamy, J.Saltz, and D.Mavriplis

October 1991

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, NY 13244-4100

DISTRIBUTED MEMORY COMPILER METHODS FOR IRREGULAR PROBLEMS – DATA COPY REUSE AND RUNTIME PARTITIONING[1]

Raja Das[a], Ravi Ponnusamy[b], Joel Saltz[a] and Dimitri Mavriplis[a]

[a]ICASE, MS 132C, NASA Langley Research Center, Hampton, VA-23666, USA

[b]Department of Computer Science, Syracuse University , Syracuse, NY 13244-4100

Abstract

This paper outlines two methods which we believe will play an important role in any distributed memory compiler able to handle sparse and unstructured problems. We describe how to link runtime partitioners to distributed memory compilers. In our scheme, programmers can *implicitly* specify how data and loop iterations are to be distributed between processors. This insulates users from having to deal explicitly with potentially complex algorithms that carry out work and data partitioning.

We also describe a viable mechanism for tracking and reusing copies of off-processor data. In many programs, several loops access the same off-processor memory locations. As long as it can be verified that the values assigned to off-processor memory locations remain unmodified, we show that we can effectively reuse stored off-processor data. We present experimental data from a 3-D unstructured Euler solver run on an iPSC/860 to demonstrate the usefulness of our methods.

[1]This work is supported by NASA contract NAS1-18605 while the authors were in residence at ICASE, NASA Langley Research Center. In addition, support for author Saltz was provided by NSF from NSF grant ASC-8819374. The authors assume all responsibility for the contents of the paper.

To appear in book - "Languages, Compilers and Runtime Environments for Distributed Memory Machines," Editors: J.Saltz and P.Mehrotra, Elsevier Press.

1 Introduction

Over the past few years, we have developed methods needed to generate efficient distributed memory code for a large class of sparse and unstructured problems. In sparse and unstructured problems, the dependency structure is determined by variable values known only at runtime. In these cases, effective use of distributed memory architectures is made possible by a runtime preprocessing phase. This preprocessing is used to partition work, to map data structures and to schedule the movement of data between the memories of processors. The code needed to carry out runtime preprocessing can be generated by a distributed memory compiler in a process we call *runtime compilation* [39].

This paper presents two new runtime compilation methods. In this paper, we describe:

how to link runtime partitioners to distributed memory compilers, and

how to reduce interprocessor communication requirements by eliminating redundant off-processor data accesses.

A *compiler-linked* runtime partitioner uses dynamic data dependency information to decompose data structures and to partition loop iterations. The compiler produces code that at runtime generates a standardized representation of the dependency graph that arises from one or more loop nests. This dependency graph representation is then passed to a compiler embedded data structure partitioner. The compiler also generates code that at runtime produces a graph that is used in a compiler embedded loop iteration partitioner. Programmers use Fortran extensions to specify which loops and which distributed arrays should be used to derive data structure partitions. Consequently, programmers *implicitly* specify how data and loop iterations are to be distributed between processors. The idea of developing a set of widely applicable partitioners has been pursued by G. Fox for many years (see for instance [15] and [16]), and a general scheme for linking such partitioners to compilers was outlined in [33]. In this paper we describe some of the runtime support and the language extensions that are allowing us to develop the software required to realize some of these ideas. In the interest of casting our vote for standardization in the development of languages and extensions for distributed memory MIMD and SIMD machines, we present our work in the context of a pre-existing language, Fortran D [17].

Once data structure and loop iteration partitioning have been determined, we carry out further preprocessing to generate communication calls needed to efficiently transport data between processors. In sparse and unstructured computations, distributed arrays are typically accessed using indirection. Runtime preprocessing is used to generate a small

number of communications calls to carry out the required data transport. In many cases, several loops access the same off-processor memory locations. As long as it is known that the values assigned to off-processor memory locations remain unmodified, it is possible to reuse stored off-processor data. A mixture of compile-time and run-time analysis can be used to recognize these situations. Compiler analysis determines when it is safe to assume that the off-processor data copy remains valid. Software primitives generate communications calls which selectively fetch only those off-processor data, which are not available locally. We will call a communications pattern that eliminates redundant off-processor data accesses an *incremental schedule.* The preprocessing described here builds on the work described in [6], [22] and [46].

We will set the context of the work in Section 2. In Section 3.1, we will describe primitives that produce incremental schedules. In Section 3.2 we will describe the primitives used to couple data and loop iteration partitioners to compilers. In Section 4 we will present an overview of our compiler effort. We describe the transformations which generate incremental inspectors and executors, and describe the language extensions we use to control compiler-linked runtime partitioning. Finally, in Section 5 we will present performance data to characterize the performance of our methods.

2 Overview

2.1 Overview of Fortran D

We will present our runtime-compilation methods in the context of Fortran D. Fortran D is a version of Fortran 77 enhanced with a rich set of data decomposition specifications, a definition of the language extensions may be found in [17]. Fortran D as currently specified requires that users explicitly define how data is to be distributed. Many researchers have explored the problem of specifying data decompositions, and FortranD has drawn extensively on this work (e.g. [46], [25], [36] and [11], [34], [7, 27, 26, 28]) 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.

Fortran D can be used to explicitly specify an irregular inter-processor partition of distributed array elements. In Figure 1, we present an example of such a Fortran D declaration. In Fortran D, one declares a template called a *distribution* 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

....

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 1: Fortran D Irregular Distribution

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, Figure 1, two size `N`, one dimensional decompositions are defined. In statement S4, decomposition `reg` is partitioned into equal sized blocks with a block assigned to each processor. In statement S5, array `map` is aligned with distribution `reg`. Array `map` will be used to specify (in statement S8) 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.

we shall illustrate in the following sections, our new language extensions and compiler techniques make it possible for programmers to *implicitly* specify how data and loop iterations are to be distributed between processors.

2.2 Overview of PARTI

In this section, we will give an overview of the functionality of the PARTI primitives described in previous publications ([46], [6], [39]). In many algorithms, data produced or input during a program's initialization plays a large role in determining the nature of the subsequent computation. In the PARTI approach, when the data structures that define a computation have been initialized, a preprocessing phase follows. Vital elements of the strategy used by the rest of the algorithm are determined by this preprocessing phase.

In distributed memory MIMD architectures, there is typically a non-trivial communications latency or startup cost. For efficiency reasons, information to be transmitted should be collected into relatively large messages. The cost of fetching array elements can be reduced by precomputing what data each processor needs to send and receive.

In irregular problems, such as solving PDEs on unstructured meshes and sparse matrix algorithms, the communication pattern depends on the input data. This typically arises due to some level of indirection in the code. In this case, it is not possible to predict at compile time what data must be prefetched. To deal with this lack of information the original sequential loop is transformed into two constructs namely, the *inspector* and the *executor*.

During program execution, the *inspector* loop examines the data references made by a processor, and calculates what off-processor data needs to be fetched and where that data will be stored once it is received. The *executor* loop then uses the information from the inspector to implement the actual computation. We have developed a suite of primitives that can be used directly by programmers to generate inspector/executor pairs.

These primitives are named PARTI (Parallel Automated Runtime Toolkit at ICASE) [12], [6]; they carry out the distribution and retrieval of globally indexed but irregularly distributed data-sets over the numerous local processor memories. Each inspector produces a set of *schedules*, which specify the communication calls needed to either:

i obtain copies of data from specified off-processor memory locations (i.e. *gather*) or,

ii modify the contents of specified off-processor memory locations (i.e. *scatter*), or

iii accumulate (e.g. add or multiply) values to specified off-processor memory locations, (i.e. *accumulate*).

Schedulers use hash tables to generate communication calls that, for each loop nest, transmit only a single copy of each off-processor datum [22], [46]. The schedules are used in the executor by PARTI primitives to gather, scatter and accumulate data to/from off-processor memory locations. In this paper, the idea of eliminating duplicates has been

taken a step further. If several loops require different but overlapping data references we can now avoid communicating redundant data (See Section 3.1 and Section 4.1.3).

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 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 be able to assign arbitrary array elements to each processor.

Each element in a distributed array is assigned to a particular processor, and in order for another processor to be able to access a given element of the array we must know the processor in which it resides, and its local address in this processor's memory. We thus build a *translation table* which, for each array element, lists the host processor address.

For a one-dimensional array of N elements, the translation table also contains N elements, and therefore must be itself be distributed over the local memories of the processors. This is accomplished by 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. If we are required to access the m^{th} element of the array, we look up its address in the distributed translation table, which we know can be found in the $(m/N) * P + 1^{th}$ processor. One of the PARTI primitives handles initialization of distributed translation tables, and other primitives are used to access the distributed translation tables.

3 The PARTI Primitives

This section describes the primitives which schedule and then carry out movement of data between processors, along with the primitives that couple partitioners to compilers. The primitives that couple partitioners to compilers are entirely new. The data movement and scheduling primitives are related to the PARTI primitives described earlier ([6] and [46]) but incorporate a number of new insights we have had about sparse and unstructured computations. These primitives differ in a number of ways from those described earlier in that the new primitives:

eliminate redundant off-processor references and

make it simple to produce parallelized loops that are virtually identical in form to the original sequential loops.

```
      real*8 x(N),y(N)
C     Loop over edges involving x, y
L1    do i=1,n_edge
        n1 = edge_list(i)
        n2 = edge_list(n_edge+i)
S1      y(n1) = y(n1) + ...x(n1) ... x(n2)
S2      y(n2) = y(n2) + ...x(n1) ... x(n2)
      end do
C     Loop over Boundary faces involving x, y
L2    do i=1,n_face
        m1 = face_list(i)
        m2 = face_list(n_face+i)
        m3 = face_list(2*n_face + i )
S3      y(m1) = y(m1) + ...x(m1) ... x(m2) ... x(m3)
S4      y(m2) = y(m2) + ...x(m1) ... x(m2) ... x(m3)
      end do
```

Figure 2: Sequential Code

To explain how the primitives work, we will use an example which is similar to loops found in unstructured computational fluid dynamics (CFD) codes. In most unstructured CFD codes, a mesh is constructed which describes an object and the physical region in which a fluid interacts with the object. Loops in fluid flow solvers sweep over this mesh structure. The two loops shown in Figure 2 represent a sweep over the edges of an unstructured mesh followed by a sweep over faces that define the boundary of the object. Since the mesh is unstructured, an indirection array has to be used to access the vertices during a loop over the edges or the boundary faces. In loop L1, a sweep is carried out over the edges of the mesh and the reference pattern is specified by integer array **edge_list**. Loop L2 represents a sweep over boundary faces, and the reference pattern is specified by **face_list**. The array **x** only appears in the right hand side of expressions in Figure 2, (statements S1 through S4), so the values of **x** are not modified by these loops. In Figure 2, array **y** is both read and written to. These references all involve accumulations in which computed quantities are added to specified elements of **y** (statements S1, S2, S3 and S4).

3.1 Primitives for Communications Scheduling

In this section we use a running example derived from Figure 2 in order to present the runtime support we need to eliminate redundant off-processor references. As was the case with our earlier suite of primitives described in [6], this runtime support can be used either by a complier or can be embedded into distributed memory codes manually by programmers. Our new primitives carry out preprocessing that make it straightforward to produce parallelized loops that are virtually identical in form to the original sequential loops. The importance of this is that it will be possible to generate the same quality object code on the nodes of the distributed memory machine as could be produced by the sequential program running on a single node.

Our primitives make use of hash tables [22] to allow us to recognize and exploit a number of situations in which a single off-processor distributed array reference is used several times. In such situations, the primitives only fetch a single copy of each unique off-processor distributed array reference.

3.1.1 PARTI Executor

Figure 3 depicts the *executor* code with embedded fortran callable PARTI procedures *dfmgather*, *dfscatter_add* and *dfscatter_addnc*. Before this code is run, we have to carry out a preprocessing phase, to be described in Section 3.1.2. This executor code changes significantly when non-incremental schedules are employed. An example of the executor

code when the preprocessing is done without using incremental schedules is given in [41].

The arrays x and y are partitioned between processors, each processor is responsible for the long term storage of specified elements of each of these arrays. The way in which x and y are to be partitioned between processors is determined by the inspector. In this example, elements of x and y are partitioned between processors in exactly the same way. Each processor is responsible for *n_on_proc* elements of x and y.

It should be noted that except for the procedure calls, the structure of the loops in Figure 3 is identical to that of the loops in Figure 2. In Figure 3, we again use arrays named x and y; in Figure 3, x and y now represent arrays defined on a single processor of a distributed memory multiprocessor. On each processor P, arrays x and y are declared to be larger than would be needed to store the number of array elements for which P is responsible. We will store copies of off-processor array elements beginning with local array elements `x(n_on_proc+1)` and `y(n_on_proc+1)`.

The PARTI subroutine calls depicted in Figure 3 move data between processors using a precomputed communication pattern. The communication pattern is specified by either a single *schedule* or by an array of schedules. *dfmgather* uses communication schedules to fetch off-processor data that will be needed either by loop L1 or by loop L2. The schedules specify the locations in distributed memory from which data is to be obtained. In Figure 3, off-processor data is obtained from array x defined on each processor. Copies of the off-processor data are placed in a buffer area beginning with `x(n_on_proc+1)`.

The PARTI procedures *dfscatter_add* and *dfscatter_addnc*, in statement S2 and S3 Figure 3, accumulate data to off-processor memory locations. Both *dfscatter_add* and *dfscatter_addnc* obtain data to be accumulated to off processor locations from a buffer area that begins with `y(n_on_proc+1)`. Off-processor data is accumulated to locations of y between indices 1 and `n_on_proc`. The distinctions between *dfscatter_add* and *dfscatter_addnc* will be described in Section 3.1.3.

In Figure 3, several data may be accumulated to a given off-processor location in loop L1 or in loop L2.

3.1.2 PARTI Inspector

In this section, we will outline how we carry out the preprocessing needed to generate the arguments needed by the code in Figure 3. This preprocessing is depicted in Figure 4.

The way in which the nodes of an irregular mesh are numbered frequently do not have a useful correspondence to the connectivity pattern of the mesh. When we partition such a mesh in a way that minimizes interprocessor communication, we may need to be able to assign arbitrary mesh points to each processor. The PARTI procedure *ifbuild_translation_table* (S1 in Figure 4) allows us to map a globally indexed distributed

```
real*8 x(n_on_proc+n_off_proc)
real*8 y(n_on_proc+n_off_proc)

S1  dfmgather(sched_array,2,x(n_on_proc+1),x)

      C  Loop over edges involving x, y
     L1  do i=1,local_n_edge
         n1 = local_edge_list(i)
         n2 = local_edge_list(local_n_edge+i)
     S1  y(n1) = y(n1) + ...x(n1) ... x(n2)
     S2  y(n2) = y(n2) + ...x(n1) ... x(n2)
         end do

S2  dfscatter_add(edge_sched,y(n_on_proc+1),y)

      C  Loop over Boundary faces involving x, y
     L2  do i=1,local_n_face
         m1 = local_face_list(i)
         m2 = local_face_list(local_n_face+i)
         m3 = local_face_list(2*local_n_face + i )
     S3  y(m1) = y(m1) + ...x(m1) ... x(m2) ... x(m3)
     S4  y(m2) = y(m2) + ...x(m1) ... x(m2) ... x(m3)
         end do

S3  dfscatter_addnc(face_sched,y(n_on_proc+1),
    buffer_mapping,y)
```

Figure 3: Parallelized Code for Each Processor

```
S1 translation_table = ifbuild_translation_table(1,myvals,n_on_proc)
S2 call flocalize(translation_table,edge_sched,part_edge_list, local_edge_list,2*n_edge,n_off_proc)
S3 sched_array(1) = edge_sched
S4 call fmlocalize(translation_table,face_sched,
     incremental_face_sched, part_face_list,local_face_list,
     4*n_face, n_off_proc_face,
     n_new_off_proc_face, buffer_mapping, 1,sched_array)
S5 sched_array(2) = incremental_face_sched
```

Figure 4: Inspector Code for Each Processor

array onto processors in an arbitrary fashion. Each processor passes the procedure *ifbuild_translation_table* a list of the array elements for which it will be responsible (**myvals** in S1, Figure 4). If a given processor needs to obtain a datum that corresponds to a particular global index i for a specific distributed array, the processor can consult the distributed translation table to find the location of that datum in distributed memory.

The PARTI procedures *flocalize* and *fmlocalize* carry out the bulk of the preprocessing needed to produce the executor code depicted in Figure 3. We will first describe *flocalize*, (S2 in Figure 4). On each processor P, flocalize is passed:

1. a pointer to a distributed translation table (translation_table in S2),
2. a list of globally indexed distributed array references for which processor P will be responsible, (**edge_list** in S2), and
3. number of globally indexed distributed array references (2*n_edge in S2).

Flocalize returns:

1. a *schedule* that can be used in PARTI gather and scatter procedures (edge_sched in S2),
2. an integer array that can be used to specify the pattern of indirection in the executor code (**local_edge_list** in S2), and
3. number of distinct off-processor references found in edge_list (n_off_proc in S2).

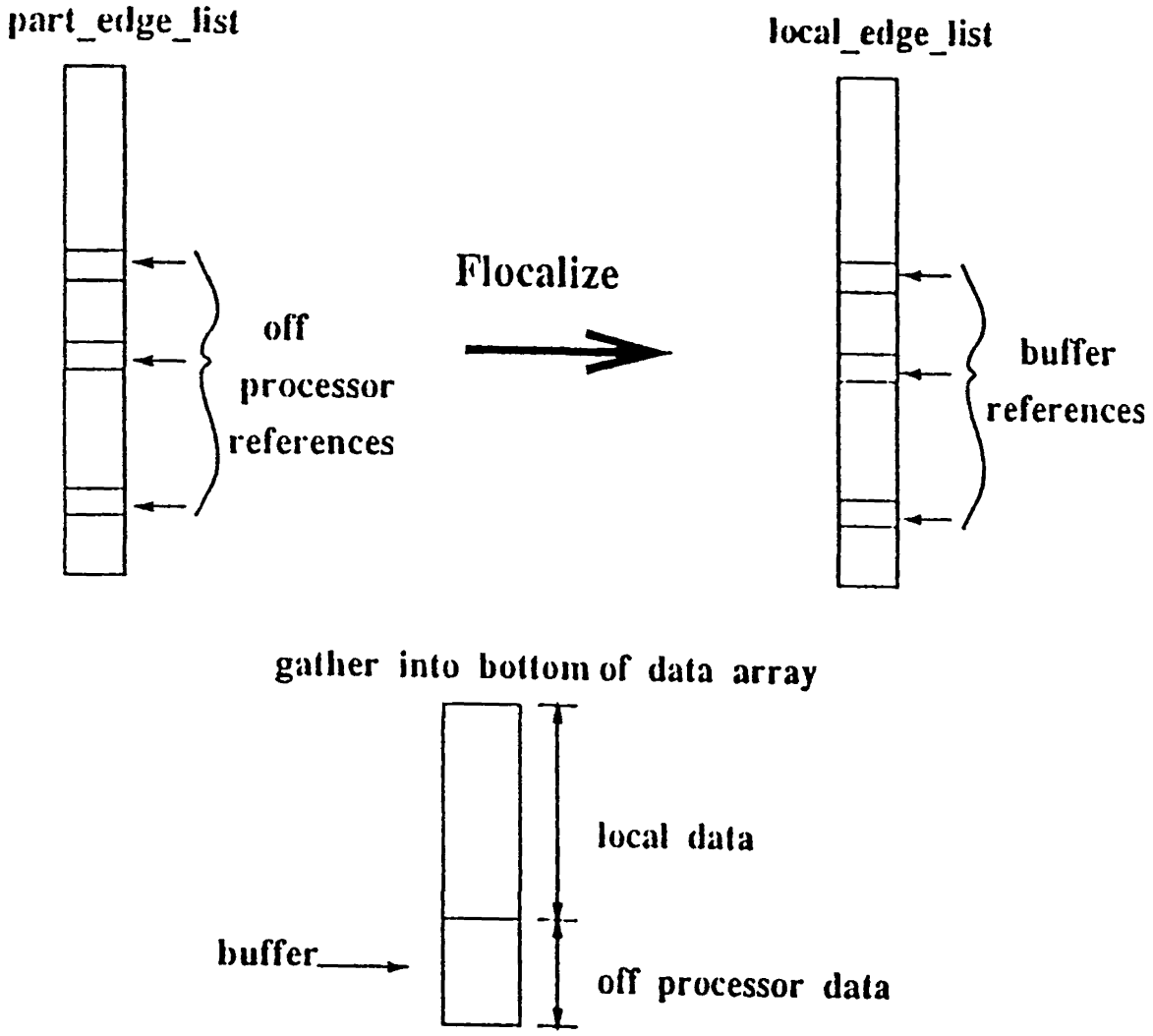

Figure 5: Flocalize Mechanism

A sketch of how the procedure *flocalize* works is shown in Figure 5. The array **edge_list** shown in Figure 2 is partitioned between processors. The **part_edge_list** passed to *flocalize* on each processor in Figure 4 is a *subset of* **edge_list** depicted in Figure 2. We cannot use **part_edge_list** to index an array on a processor as **part_edge_list** refers to globally indexed elements of arrays **x** and **y**. *Flocalize* changes this **part_edge_list** so that valid references are generated when the edge loop is executed. The buffer for each data array is placed immediately following the on-processor data for that array. For example, the buffer for data array **x** starts at **x(n_on_proc+1)**. Hence, when *flocalize* changes the **part_edge_list** to **local_edge_list**, the off-processor references are changed to point to the buffer addresses. When the off processor data is collected into the buffer using the *schedule* returned by *flocalize*, the data is stored in a way such that execution of the edge loop using the **local_edge_list** accesses the correct data.

There are a variety of situations in which the same data need to be accessed by multiple loops (Figure 2). In Figure 2, no assignments to **x** are carried out. In the beginning of Figure 3, each processor can gather a single copy of every distinct off-processor value of **x** referenced by loops L1 or L2. The PARTI procedure *fmlocalize* (S4 in Figure 4) makes it simple to remove these duplicate references. *fmlocalize* makes it possible to obtain only those off-processor data not requested by a given set of pre-existing schedules. The procedure *dfmgather* in the executor in Figure 3 obtains off-processor data using *two schedules*; *edge_sched* produced by *flocalize* (S2 Figure 4) and *incremental_face_sched* produced by *fmlocalize* (S4 Figure 4).

The pictorial representation of the *incremental schedule* is given in Figure 6. The schedule to bring in the off-processor data for the edge_loop is given by the *edge schedule* and is formed first. During the formation of the schedule to bring in the off-processor data for the face_loop we remove the duplicates shown by the shaded region in Figure 6. Removal of duplicates is achieved by using a hash table. The off-processor data to be accessed by the edge schedule is first hashed using a simple hash function. Next all the data to be accessed during the face_loop is hashed. At this point the information that exists in the hash table allows us to remove all the duplicates and form the *incremental schedule*. In Section 5 we will present results showing the usefulness of incremental schedule.

To review the work carried out by fmlocalize, we will summarize the significance of all but one of the arguments of this PARTI procedure. On each processor, fmlocalize is passed:

1. a pointer to a distributed translation table (translation_table in S4),

2. a list of globally indexed distributed array references. (face_list in S4),

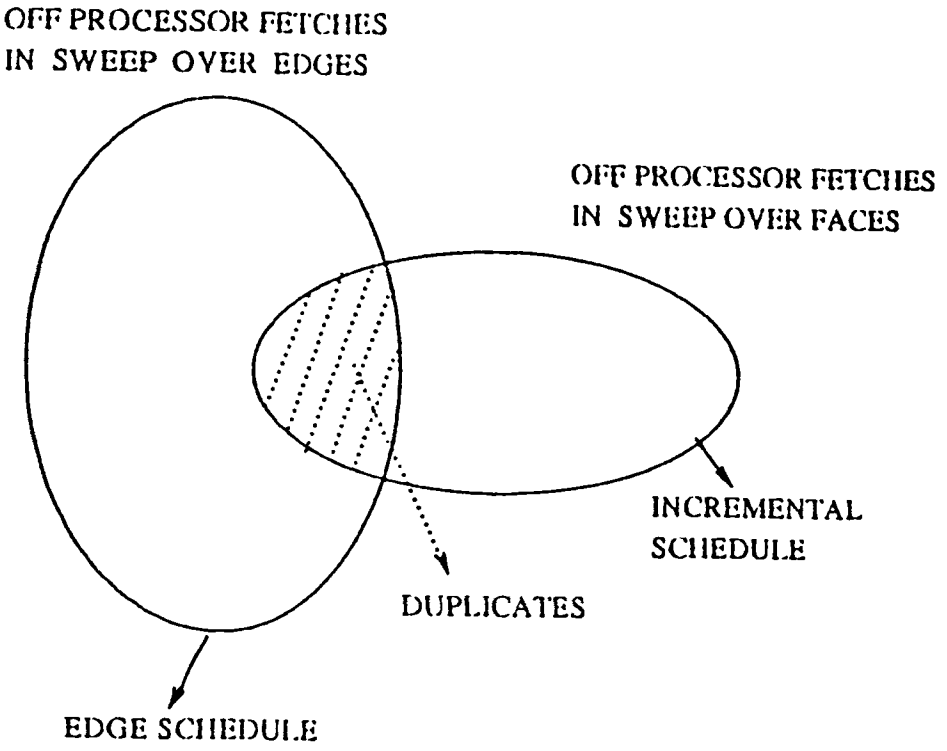

Figure 6: Incremental Schedule

3. number of globally indexed distributed array references (4*n_face in S4),
4. number of pre-existing schedules that need to be taken into account when removing duplicates (1 in S4), and
5. an array of pointers to pre-existing schedules (sched_array in S4).

Fmlocalize returns:

1. a *schedule* that can be used in PARTI gather and scatter procedures. This schedule *does not take any pre-existing schedules into account* (face_sched in S4),
2. an *incremental schedule* that includes only off-processor data accesses not included in the pre-existing schedules (incremental_face_sched in S4),
3. a list of integers that can be used to specify the pattern of indirection in the executor code (local_face_list in S4),
4. number of distinct off-processor references in face_list (n_off_proc_face in S4).
5. number of distinct off-processor references not encountered in any other schedule (n_new_off_proc_face in S4).
6. buffer_mapping - to be discussed in Section 3.1.3.

3.1.3 A Return to the Executor

We have already discussed *dfmgather* in Section 3.1.1 but we have not said anything so far about the distinction between *dfscatter_add* and *dfscatter_addnc*. When we make use of incremental schedules, we assign a single buffer location to each off-processor distributed array element. In our example, we carry out separate off-processor accumulations after loops L1 and L2. As we will describe below, in this situation, our off-processor accumulation procedures may no longer reference consecutive elements of a buffer.

We assign copies of distinct off-processor elements of **y** to buffer locations, to handle off-processor accesses in loop L1, Figure 3. We can then use a schedule (edge_sched) to specify where in distributed memory each consecutive value in the buffer is to be accumulated. PARTI procedure *dfscatter_add* can be employed; this procedure uses schedule *edge_sched* to accumulate to off-processor locations consecutive buffer locations beginning with **y(n_on_proc + 1)**. When we assign off-processor elements of **y** to buffer locations in L2, some of the off-processor copies may already be associated with buffer locations. Consequently in S3, Figure 3, our schedule (face_sched) must access buffer locations in an irregular manner. The pattern of buffer locations accessed is specified by integer array *buffer_mapping* passed to *dfscatter_addnc* in S3, Figure 3. (*dfscatter_addnc* stands for dfscatter_add non-contiguous)

3.2 Mapper Coupler

In irregular problems, it is frequently desirable to allocate computational work to processors by assigning all computations that involve a given loop iteration to a single processor [6]. We consequently partition *both* distributed arrays and loop iterations. Our approach is to first partition distributed arrays and then, based on distributed array partitionings, partition loop iterations. This appears to be a practical approach as in many cases, the same set of distributed arrays are used by many loops.

When we partition distributed arrays, we have not yet assigned loop iterations to processors. We do 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.

There are many partitioning methods available, [42], [15], [5] [9] but currently partitioners must be coupled to user programs in a manual fashion. This manual coupling is particularly troublesome when we wish to make use of parallelized partitioners. Here, we introduce a new notion of linking partitioners with programs by producing a generic

data structure at run time, which is independent of the problems. For this purpose, we have developed primitives which can generate a standardized input format for the partitioners. In our approach the standardized data structure is generated from the loops in the problem specified by users using certain language extensions to be discussed in Section 4.1.

We now outline what needs to be done to link a data partitioner with a program in which a specific loop has been specified using the language extensions described in Section 4.1. We first consider loops in which all distributed arrays appearing in a loop conform in size and are to be distributed in an identical manner. We also restrict ourselves to loops without loop carried dependencies (this restriction will be relaxed slightly later in this section). We define a statement *bipartite runtime dependency graph* (statement BRDG) to represent the dependencies between the index of a distributed array element defined on the left hand side of a loop statement **S** and the indices of all distributed array elements on the right hand side of **S**. As the name implies, statement BRDG is a bipartite directed graph. We merge the statement BRDG associated with each statement **S** in a loop to form a *loop* **BRDG**. When we merge l links we associate a weight l with the merged vertex. Most data partitioners make use of undirected connectivity graphs. When all distributed arrays appearing in a statement conform, we can collapse the loop **BRDG** into a undirected graph, the loop *runtime dependence graph* or the loop **RDG**. The weight associates with each edge of the loop **RDG** is the sum of the weights of the two collapsed **BRDG** edges.

A loop **RDG** is constructed by adding edge $< i, j >$ between nodes i and j either when

> a reference to array index i appears on the left side of an expression and a reference to j appears on the right side, or
>
> a reference to array index j appears on the left side of an expression and a reference to i appears on the right side.

Each time edge $< i, j >$ is encountered, we increment a counter associated with $< i, j >$. Accumulation type output dependency edges of type $< i, i >$ are ignored in the graph generation process as the presence of such dependencies do not induce inter-processor communication. The loop **RDG** is currently represented by a distributed data structure [31], this data structure is closely related to Saad's Compressed Sparse Row (CSR) format (see [38]).

Data partitioning is carried out as follows. We assume **P** processors.

1. At *compile time* a *dependency coupling code* is generated. This code produces a loop **RDG** at runtime,

2. The loop **RDG** is passed to a *data partitioning* procedure that partitions the loop **RDG** into **P** subgraphs. The **RDG** vertices assigned to each subgraph correspond to a distributed array distribution.

3. The output of the partitioning procedure is a distributed translation table. This translation table is associated with each of the identically distributed arrays referenced in the loop.

Once we have partitioned data, we must partition computational work. One convention is to compute a program statement S in the processor associated with S's left hand side distributed array element. (If the left hand side of **S** references a replicated variable then the work is carried out in all processors). Were we to assign work in this manner, we would want to partition the **RDG** for statement **S** in a way that would correspond to reducing the combined cost of load imbalance and the cost of interprocessor communication. Each **RDG** edge to cross a boundary between partitions would correspond to either a unidirectional or bidirectional data communication. Instead of assigning work to the processor associated with S's left hand distributed array element, we partition distributed arrays and loop iterations separately. Our motivation for using the loop **RDG** as an input to a data partitioner comes from our decision to attempt to partition loop iterations so as to minimize off-processor distributed array references.

To partition loop iterations, we use 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. The **RIG** is generated for every loop that references at least one irregularly distributed array. The *runtime iteration processor assignment graph* or **RIPA** lists, for each loop iteration, the number of distinct data references associated with each processor.

We partition loop iterations in the following manner:

1. The **RIG** is generated for each loop in which distributed arrays are referenced.

2. The processor assignment is found for for each distributed array reference appearing in a **RIG**. If the distributed array is irregularly distributed, this information is obtained using the array's distributed translation table (Section 2.2). The processor assignments are used to generate the **RIPA** graph.

3. Loop iterations are partitioned using an *iteration partitioning procedure* which makes use of the **RIPA** graph.

Just as there are many possible strategies that can be used to partition data, there are also many strategies that could be used to partition loop iterations. We currently

employ strategies that assign loop iterations to the processor associated with the largest number of distributed array references in the **RIG**.

3.3 Compiler-linked Mapping: Runtime Support

In this section we outline the primitives employed to carry out compiler-linked data and loop iteration partitioning.

We begin each compiler-linked mapping with an initial distribution of loop iterations and of integer indirection arrays needed to determine distributed array references. The object of this initial preprocessing is to extract information needed for mapping. In many cases, the initial distribution of loop iterations I_{init}, will be a simple default distribution. In some situations (e.g. adaptive codes), preprocessing to support irregular array mappings may have already been carried out. Thus integer indirection arrays may have already been irregularly distributed when we begin our derivation of a compiler-linked mapping. Our runtime support will handle either regular or irregular initial loop iteration distributions I_{init}. The *local* loop **RDG** is defined as the restriction of the loop **RDG** to a single processor. The local loop **RDG** includes only distributed array elements associated with I_{init}.

Procedure *eliminate_dup_edges* uses a hash table to store unique directed dependency edges, along with a count of the number of times each edge has been encountered. Once all edges in a loop have been recorded, **edges_to_RDG** generates the local loop **RDG** and then merges all local loop **RDG** graphs to form the loop **RDG**. The data structures that describe the loop **RDG** graph are passed to a data partitioner *RDG_partitioner*. *RDG_partitioner*. returns a pointer to a distributed translation table that describes the new mapping. Note that *RDG_partitioner*. can use any heuristic to partition the data, the only constraint is that the partitioners have the correct calling sequence.

We consider the sequential code depicted in Figure 2 to illustrate how the primitives can be used to link partitioners with programs. We assume that the user has specified using the language extensions that arrays **x** and **y** are to be partitioned based the loop L1 in a conforming manner. At *compile time*, a sequence of calls to a set of mapper coupler primitives are embedded as shown in Figure 7.

The partitioning of loop iterations is supported by two primitives, *deref_rig* and *partition_rig*. The **RIG** is generated by code transformed by a compiler. The primitive *deref_rig* inputs the **RIG**. This primitive accesses distributed translation tables to find the processor assignments associated with each distributed array reference. *deref_rig* returns the **RIPA**. The **RIPA** is partitioned using the iteration partitioning procedure, *iter_partition*.

partition loop iterations between processors in blocks

partition integer indirection array **edge_list** so that if iteration i is assigned to processor **P**, **edge_list(i)** and **edge_list(n_edge+i)** are on **P** (methods needed to carry out this preprocessing are described in [46]).

do i=1,n_edge

pass dependency edges (n1,n2), (n2,n1) to procedure *eliminate_dup_edges*

end do

obtain loop **RDG** data structure from hash table using procedure *edges_to_RDG*

loop **RDG** is passed to *RDG_partitioner*. A pointer is returned to a distributed translation table which describes the new mapping.

Figure 7: Runtime Support for Deriving Irregular Data Distributions

4 PARTI Compiler

In this section we first describe language extensions which allow a programmer to implicitly specify how data and loop iterations are to be partitioned between processors. We then outline compiler transformations used to carry out this implicitly defined work and data mapping. The compiler transformations generate code which embeds the mapper coupler primitives described in Section 3.2. In addition we outline compiler transformations needed to take advantage of the incremental scheduling primitives described in Section 3.1.

4.1 Compiler-Linked Problem Mapping

4.1.1 Overview

The current Fortran D syntax outlined in Section 2.1 requires programmers to *explicitly* define irregular data decompositions.

In Figure 1, we align real arrays `x` and `y` with the decomposition *irreg* (statement S5). The array `map` is used to specify the distribution of `irreg`. Integer array `map` is aligned with decomposition *reg* (statement S4) and then *reg* is distributed by amoung the processors blocks (statement S6). The meaning of the statement S7 is that the distribution of decomposition *irreg* is determined by *values* assigned to `map`. For example, if the value map(100) is 10, this indicates that both x(100) and y(100) are assigned to processor 10.

The difficulty with the declarations depicted in Figure 1 is that *it is not obvious how one would partition the irregularly distribute 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 (see for instance [42], [15], [5]), coding such partitioners from scratch can represent a significant effort. There is no standard interface between the partitioners and the different problems. The partitioners described in the literature typically operate on data structures whose physical interpretation is known to the programmer (e.g. meshes in finite difference equations, sparse matrices in sparse linear systems solvers, etc).

Our approach is to identify a nest of loops `L` that involves each irregularly distributed array we will need to partition. From the loop `L`, we produce at compile time a mapper coupler (see Section 3.2)

Figure 8 is derived from the sequential code in Figure 2. The code in Figure 8 contains loops L1 and L2 from the code in Figure 2. To simplify presentation, only L1 is depicted explicitly in Figure 8.

We use statement S4 to designate loop `L1` as the loop that will be used to generate a mapper coupler. **implicitmap**(x,y) indicates that an **RDG** graph is to be generated based on the dependency relations between distributed arrays `x` and `y` in loop `L1`. We assume that all arrays listed in an implicitmap statement are to be identically distributed and that the loop in question parallelizes, except for possible accumulation type output dependencies (If the compiler cannot determine that these assumptions are valid, an error is reported).

In many codes used to solve mesh based problems, we can specify a nest of loops so that the **RDG** will represent the original mesh. For instance, in Figure 8, loop L1 represents a sweep over the edges of a mesh. The **RDG** obtained from statement S4 recaptures the original mesh topology.

It is easy to generalize the language extensions described here so that we specify an implicit mapping using more than one loop. In this case, a multiple loop **RDG** is generated based on merged dependency patterns arising from the loops.

4.1.2 Embedding Mapper Coupler Primitives

We use the example in Figure 8 to show how the compiler primitives are embedded in the code. When the statement **distribute ... implicit using** is encountered in the code, the compiler locates the loop L specified by the user. The indirection pattern in this loop will be used to generate the **RDG**. In order for the executable statement **distribute ... implicit using** to make sense, we must be able to anticipate how the distributed arrays

```
    ....
    real*8 x(N),y(N)
    decomposition coupling(N)
S1  if(remap.eq.yes) then
S2  distribute coupling(implicit using edges)
    endif
S3  align x,y with coupling
    ....
S4  implicitmap(x,y) edges
C   Loop over edges involving x, y
L1  do i=1,n_edge
    n1 = edge_list(i)
    n2 = edge_list(n_edge+i)
S1  y(n1) = y(n1) + ...x(n1) ... x(n2)
S2  y(n2) = y(n2) + ...x(n1) ... x(n2)
    end do
    ....
    L2 Loop over faces involving x, y
```

Figure 8: Example of Implicit Mapping

in L will be indexed. when L is next encountered. We need to be able to determine that all relevant reference patterns in L can be predicted when **distribute ... implicit using** executes. In our simple example (Figure 8), the implicit distribution statement is located in the same procedure as the user specified loop. The compiler must identify all variables V that determine the subscript functions of distributed arrays in L and must determine whether there is any chance that any members of V could be killed between **distribute ... implicit using** and loop L. In this case, standard data flow analysis can determine whether any assignment has been made to a member of V. In many cases, the implicit distribution statement might not be placed in the same procedure as L. In this case, we will require the results of interprocedural analysis.

When L is identified and indexing information pertaining to L is obtained, a transformed loop L' is generated. L' contains the calls to *eliminate_dup_edges* that will be needed to generate the local loop **RDG** (see Section 3.2). Recall from Section 3.2 that *eliminate_dup_edges* produces a hash table. A pointer to this hash table is passed to procedure *edges_to_RDG*. This procedure produces a loop **RDG** which is passed to a data partitioner, *RDG_partitioner*.

Loop iterations are partitioned at runtime whenever a loop accesses at least one irregularly distributed array. Corresponding to each such loop L is generated a loop L" which generates the **RIG**. As described in Section 3.2, the **RIG** is passed to *deref_rig* to produce the **RIPA**. The **RIPA** in turn is partitioned using the iteration partitioning procedure, *iter_partition*.

4.1.3 Inspector/Executor Generation for Incremental Scheduling

Inspectors and executors must be generated for loops in which distributed arrays are accessed via indirection. Inspectors and executors are also needed in most loops that access irregularly distributed arrays. In this section we outline what must be done to generate distributed memory programs which make effective use of incremental and non-incremental schedules. Most of what we describe is as yet unimplemented, although we have constructed and benchmarked a simple compiler capable of carrying out local transformations to embed non-incremental schedules. This work is described in [46].

We first outline what must be done to generate an inspector and an executor for a program loop L. We assume that dependency analysis has determined that L either has no loop carried dependencies, or has only the simple accumulation type output dependencies of the sort exemplified in Figure 2. It should be noted that the calling sequences of the compiler-embeddable PARTI primitives differ somewhat from the primitives described in Section 3. The functionality described in primitives *flocalize* and *fmlocalize* are each implemented as a larger set of simpler primitives.

We scan through the loop and find the set of distributed arrays A that are irregularly distributed or are indexed using indirection. Information needed to generate a schedule for a given distributed array reference, can be produced from the subscript function of the reference along with knowledge of an array's distribution. We must check to make sure that the the subscript functions of all members of A are loop invariant as the methods described in this paper do not address cases in which indexing patterns are modified by computations carried out within the loop. As long as a distributed array's indexing pattern is not modified by computations carried out within a loop, a compiler can generate preprocessing code that can be hoisted out of L. This preprocessing code produces a representation of the distributed array's indexing pattern. For instance, consider the following loop:

```
do i=1,n
n1 = nde(2*i)
n2 = nde(2*i-1)
.. = x(n1) ... y(n2)
.. = ... z(n2)
end do
```

The subscript function of **y** and **z** (using notation from the Fortran 90 array extensions) is nde(2:2*n:2), and the subscript function of **x** is nde(1:2*n-1:2). Recall from Section 2.2, that schedules specify communication *patterns* and are not bound to a specific distributed array. We can avoid having to compute redundant schedules when we know that the same communication pattern will reoccur in more than one place in a loop. For instance, if **y** and **z** in the above loop are partitioned in a conforming manner, we need only to compute a single schedule to bring in off-processor elements of **y** and **z**.

Optimizations that reduce the number of schedules reduce the preprocessing time required by the *inspector*. Obviously, the elimination of redundant schedules also has a favorable impact on storage requirements. Minor modifications of common subexpression elimination should be reasonably effective in identifying redundant schedules. In [46] we describe a compiler that carries out this optimization in a rudimentary manner.

The use of incremental schedules, (Section 3.1), make it possible to avoid retransmission of unchanged distributed array. As we will show in Section 5, proper use of incremental schedules can have a marked effect on the time spent on communication. In order to make use of previously stored copies of distributed array elements, we must

ensure that the off-processor copies are still valid. Recall that we assumed that loop L had no loop carried dependencies. Thus our decision to assign each loop iteration to a single processor ensures that off-processor data obtained immediately before entering L will continue to be valid in L. The generation of incremental schedules can be carried out in two passes. A compiler first generates an inspector and executor for L with full schedules. During the second pass, some full schedules will be replaced with incremental schedules. In order to replace a full schedule with an incremental schedule, we need to know which schedules will have already caused the storage of off-processor data within L.

Generation of efficient inspectors and executors for loop L requires us to obtain information about a program as a whole. When L is called multiple times we attempt to reuse previously computed schedules. Each time L is called, we need to determine whether it is possible that the subscript functions or loop distributions in the set of distributed arrays A have been modified since the last call to L.

Analysis must also be carried out if we are to use incremental schedules to eliminate duplicate data communications between loops. We need rather comprehensive information about the program behavior. Consider a right hand side reference to distributed array **x** in program statement S for which we would like to use an incremental schedule. We will need to know

> when off-processor data copies of values of **x** become invalidated by new assignments, and
>
> which communications schedules will have already been invoked by the time we reach **x**,

Methods exist which appear likely to allow us to be able to do a reasonable job of achieving both of these objectives for many irregular scientific codes. A program dependence graph (e.g. [13], [10]) is a directed graph whose vertices represent the assignment statements and control predicates that occur in a program. The edges represent *dependences* among program components. An edge represents either a control dependence or a data dependence. Each time a schedule is used, new copies of off-processor array elements become available. In order to generate an incremental schedule for **x** at S, we need to know which schedules have already caused the storage of potentially reusable off-processor data. We can view this off-processor data reuse as a type of dependence and represent this dependence as a specific type of edge in a program dependence graph. We will call this kind of dependence edge a *reuse* edge. Using *slicing* methods, [44], [23] we can find all statements and predicates of a program that might affect the values of the distributed array reference to **x** in statement S. In ongoing joint work with Kennedy's

group at Rice, we are currently developing a variant of slicing methods which will allow us to automate the use of incremental schedules. The results of this work will be implemented as part of the Fortran D compiler being developed at Rice [21].

5 Experimental Results

5.1 Timing Results from the Euler Equation Solver

The PARTI procedures described in Section 3 were used to port a 3-D unstructured mesh Euler solver [32]. The Euler code iterates until it has computed a steady state solution on a given mesh. Two versions of the Euler solver were tested, one version used the primitive *flocalize* and *fmlocalize* to generate incremental schedules (Section 3.1), the other used only the primitive *flocalize* and generated only non-incremental schedules (Section 3.1.2). The 3-D unstructured mesh Euler solver was tested using a sequence of structurally similar meshes of varying sizes. The smallest mesh used had 3.6K vertices, the largest mesh had 210K vertices and 1.2 million edges. Figure 9 depicts a surface view of the 210K vertex mesh. The unstructured meshes were partitioned by the method described in [42].

Table 1 shows the timings obtained using non-incremental communication scheduling and Table 2 were obtained using *incremental schedules.* The single node code for these meshes run at approximately 4 Megaflops. We conjecture that the single node performance is relatively poor because the data access patterns in unstructured mesh computations are highly irregular and the number of memory references per floating point operation is very high. Both of these characteristics make it difficult for the Intel 80860 architecture to keep the processor supplied with data.

The use of incremental scheduling had a significant impact on communications costs. For instance, on the 26K mesh, the communications cost per iteration on 16 processors was 2.0 seconds when we did not employ incremental schedules. The communications cost dropped to 1.1 seconds when we used incremental schedules. On the 210K mesh on 64 processors the communications cost per iteration dropped from 3.7 seconds to 2.3 seconds when we employed incremental schedules.

Since the form of the sequential code and the parallelized code is virtually identical, we did not expect the parallelization process to introduce any new inefficiencies beyond those exacted by the preprocessing and by the calls to the primitives. We compared the parallel code running on a single node with the sequential code and found only a 2 % performance degradation. In the parallelized Euler codes, the total cost of all preprocessing was insignificant compared to the execution times required to solve the

Size Mesh	Number of Processors					
		1	2	8	16	64
3.6K	Mflops	4.1	6.0	12.0	14.4	-
	Time/iter(s)	4.6	3.1	1.5	1.3	-
	comm/iter(s)	-	0.5	0.9	0.9	-
26K	Mflops	-	-	19.2	29.9	
	Time/iter(s)	-	-	7.1	4.5	
	comm/iter(s)	-	-	2.3	2.0	
210K	Mflops	-	-	-	-	118.6
	Time/iter(s)	-	-	-	-	8.4
	comm/iter(s)	-	-	-	-	3.7

Table 1: Timings from Intel iPSC/860 : Unstructured Mesh Using Non-Incremental Schedule

problems. The program typically requires at least 100 iterations to converge, and the preprocessing times were less than 3 % of the parallel execution times.

5.2 Timing Results using the Mapper Coupler

In this section, we present data that indicates that the costs incurred by the mapper coupler primitives were roughly on the order of the cost of a single iteration of our unstructured mesh code. We also show that the mapper coupler costs are quite small compared to the cost of partitioning the data. In Table 3, *graph generation* depicts the time required by the mapper interface to generate the runtime dependence graph (**RDG**) data structure (Section 3.2. These timings involve a loop over edges that is functionally equivalent to loop L1 in Figure 2. The graph generation time includes the time required to call *eliminate_dup_edges* and the time required to call *edges_to_RDG* (Section 3.3)

In Table 3, *mapper* depicts the time needed to partition the **RDG** using using a parallelized version of Simon's eigenvalue partitioner [42]. We partitioned the **RDG** into a number of subgraphs equal to the number of processors employed. The cost of the partitioner is relatively high both because of the partitioner's high operation count and because only a modest effort was made to produce an efficient parallel implementation. It should be noted that any parallelized graph partitioner could be used as a mapper. The *iter partitioner* time shown in Table 3 gives the time needed to partition loop iterations among processors. The table also includes the time needed for a single iteration of the Euler code for different problem sizes.

Size Mesh		Number of Processors				
		1	2	8	16	64
3.6K	Mflops	4.1	7.1	16.9	17.4	-
	Time/iter(s)	4.6	2.6	1.1	1.1	-
	comm/iter(s)	-	0.3	0.5	0.7	-
26K	Mflops	-	-	23.8	38.8	
	Time/iter(s)	-	-	5.6	3.4	
	comm/iter(s)	-	-	1.1	1.1	
210K	Mflops	-	-	-	-	144.3
	Time/iter(s)	-	-	-	-	7.1
	comm/iter(s)	-	-	-	-	2.3

Table 2: Timings from Intel iPSC/860 : Unstructured Mesh Using Incremental Schedule

Table 3: Mapper Coupler Timings from Intel iPSC/860

Number of Vertices		Number of Processors					
		2	4	8	16	32	64
3.6K	graph generation (secs.)	0.34	0.24	0.21	0.20	-	-
	mapper (secs)	15.92	11.50	12.11	14.92	-	-
	iter partitioner (secs)	0.94	0.57	0.42	0.34	-	-
	comp/iter (secs)	2.4	1.31	0.6	0.34	-	-
9.4K	graph generation (secs.)	-	0.86	0.69	0.53	0.35	-
	mapper (secs)	-	70.96	62.3	65.2	89.7	-
	iter partitioner (secs)	-	1.19	0.82	0.60	0.43	-
	comp/iter(secs)	-	4.83	2.35	1.1	0.67	-
54K	graph generation (secs.)	-	-	-	-	1.50	0.94
	mapper (secs)	-	-	-	-	544.81	673.14
	iter partitioner (secs)	-	-	-	-	3.30	3.03
	comp/iter(secs)	-	-	-	-	6.06	3.81

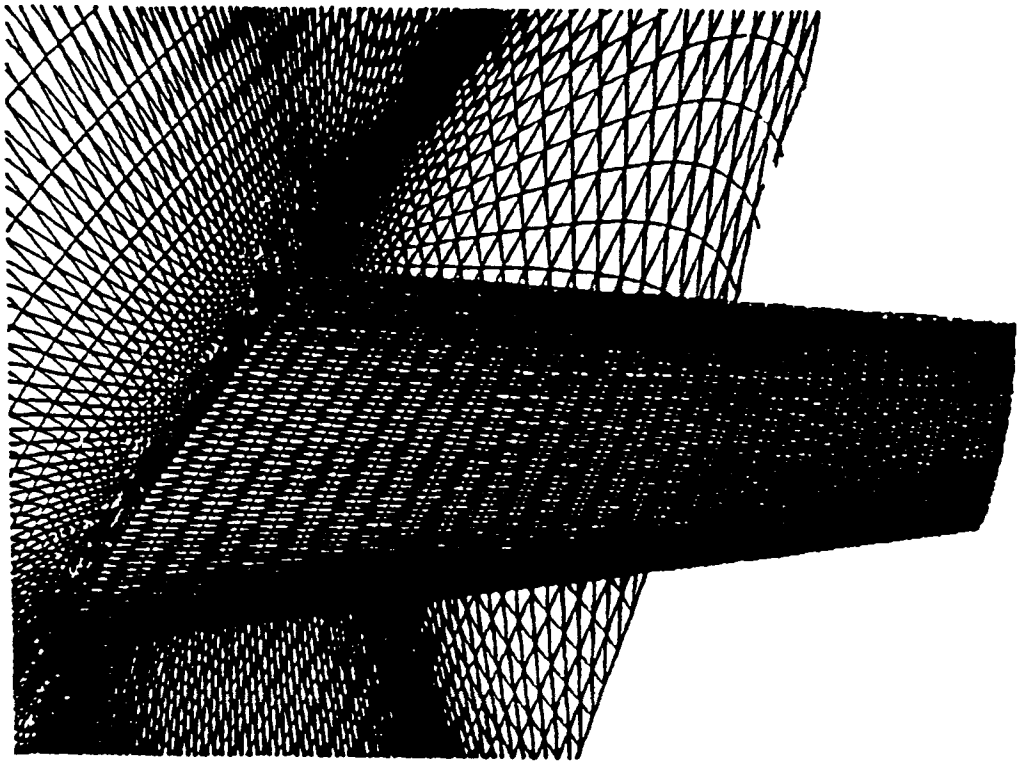

Figure 9: Surface View of Unstructured Mesh Employed for Computing Flow over ONERA M6 Wing , Number of nodes = 210K

6 Conclusions

Programs designed to carry out a range of irregular computations including sparse direct and iterative methods require many of the optimizations described in this paper. Some examples of such programs are described in [2], [29], [4], [45] and [18].

Several researchers have developed programming environments that are targeted towards particular classes of irregular or adaptive problems. Williams [45] describes a programming environment (DIME) for calculations with unstructured triangular meshes using distributed memory machines. Baden [3] has developed a programming environment targeted towards particle computations. This programming environment provides facilities that support dynamic load balancing. DecTool [9] is an interactive environment designed to provide facilities for either automatic or manual decompositions of 2-D or 3-D discrete domains.

There are a variety of compiler projects targeted at distributed memory multiprocessors [47], [8], [37], [35], [1], [43], [14], [19], [20], [24], [7, 27, 26, 28] [25], [21], [46]. Runtime compilation methods are employed in four of these projects; the Fortran D project [21], the Kali project [25], Marina Chen's work at Yale [30] and our PARTI project [33], [40], [46], and [39]. The Kali compiler which was the first compiler to implement inspector/executor type runtime preprocessing [25] and the ARF compiler which was the first compiler to support irregularly distributed arrays [46]. In related work, Lu and Chen have reported some encouraging results on the potential for effective runtime parallelization of loops in distributed memory architectures [30].

This paper has presented two new runtime compilation methods, and described in detail the required runtime support. We described how to design distributed memory compilers capable of carrying out dynamic workload and data partitions. We also described how to reduce interprocessor communication requirements by eliminating redundant off-processor data accesses. This runtime support required for this methods has been implemented in the form of PARTI primitives. We first described the design of the PARTI primitives, and then outlined the compiler transformations that embed these primitives.

We implemented a full unstructured mesh computational fluid dynamics code by embedding our runtime support by hand and have presented our performance results. These performance results demonstrated that our method for eliminating redundant off-processor communication had a significant impact on communications costs. Our performance results also demonstrated that the costs incurred by the mapper coupler primitives were roughly on the order of the cost of a single iteration of our unstructured mesh code and were quite small compared to the cost of the partitioner itself. We did not compare the time required by the PARTI primitives to Intel send and receive calls in this paper. In [6] we presented such a comparison and found that overheads incurred by using PARTI appear to be quite modest (no more than 20 %).

We have joined forces with the Fortran D group in compiler development and are implementing the methods described in this paper in the context of Fortran D in cooperation with Kennedy's group at Rice.

The non-incremental PARTI primitives described in Section 3.1 are available for public distribution and can be obtained from netlib or from the anonymous ftp cite ra.cs.yale.edu. The incremental PARTI primitives and the Mapper coupler primitives described in Section 3.2 will be released soon and will be available through the same sources..

Acknowledgements

The authors would like to thank Geoffrey Fox for many enlightening discussions about universally applicable partitioners; we would also like to thank Ken Kennedy, Chuck Koelbel and Seema Hiranandani for many useful discussions about integrating into Fortran-D runtime support for irregular problems.

The authors would also like to thank: Dennis Gannon for the use of his Faust system and his help in getting us started with Faust, Horst Simon for the use of his unstructured mesh partitioning software; and Venkatakrishnan for useful suggestions for low level communications scheduling.

References

[1] F. André, J.-L. Pazat, and H. Thomas. PANDORE: A system to manage data distribution. In *International Conference on Supercomputing*, pages 380–388, June 1990.

[2] C. Ashcraft, S. C. Eisenstat, and J. W. H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SISSC*, 11(3):593–599, 1990.

[3] S. Baden. Programming abstractions for dynamically partitioning and coordinating localized scientific calculations running on multiprocessors. *To appear, SIAM J. Sci. and Stat. Computation.*, 1991.

[4] D. Baxter, J. Saltz, M. Schultz, S. Eisentstat, and K. Crowley. An experimental study of methods for parallel preconditioned krylov methods. In *Proceedings of the 1988 Hypercube Multiprocessor Conference, Pasadena CA*, pages 1698,1711, January 1988.

[5] 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.

[6] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory architectures. *Concurrency: Practice and Experience*, 3(3):159–178, June 1991.

[7] M. C. Chen. A parallel language and its compilation to mulitprocessor archietctures or vlsi. In *2nd ACM Symposium on Principles Programming Languages*, January 1986.

[8] A. Cheung and A. P. Reeves. The paragon multicomputer environment: A first implementation. Technical Report EE-CEG-89-9, Cornell University Computer Engineering Group, Cornell University School of Electrical Engineering, july 1989.

[9] N.P. Chrisochoides, C. E. Houstis, E.N. Houstis, P.N. Papachiou, S.K. Kortesis, and J.R. Rice. Domain decomposer: A software tool for mapping pde computations to parallel architectures. Report CSD-TR-1025, Purdue University, Computer Science Department, September 1990.

[10] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Not. 23,7*, pages 57–66, July 1988.

[11] Thinking Machines Corporation. CM Fortran reference manual. Technical Report version 1.0, Thinking Machines Corporation, Feb 1991.

[12] R. Das, J. Saltz, and H. Berryman. A manual for parti runtime primitives - revision 1 (document and parti software available through netlib). Interim Report 91-17, ICASE, 1991.

[13] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 1987.

[14] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1990.

[15] G. Fox. A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube. In *The IMA Volumes in Mathematics and its Applications. Volume 13: Numerical Algorithms for Modern Parallel Computer Architectures Martin Schultz Editor.* Springer-Verlag, 1988.

[16] G. Fox and W. Furmanski. Load balancing loosely synchronous problems with a neural network. In *Third Conf. on Hypercube Concurrent Computers and Applications*, pages 241–27278, 1988.

[17] 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.

[18] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Computers.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[19] H. M. Gerndt. Automatic parallelization for distributed memory multiprocessing systems. Report ACPC/ TR 90-1, Austrian Center for Parallel Computation, 1990.

[20] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and J. Anderson. A production quality C* compiler for hypercube machines. In *3rd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 73–82, April 1991.

[21] 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.

[22] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing, to appear*, 12, August 1991.

[23] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, January 1990.

[24] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *DMCC5*, pages 1105–1114, Charleston, SC, April 1990.

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

[26] J. Li and M. Chen. Generating explicit communication from shared-memory program references. In *Proceedings Supercomputing '90*, November 1990.

[27] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-reference between distributed arrays. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Computation*, October 1990.

[28] J. Li and M. Chen. Automating the coordination of interprocessor communication. In *Programming Languages and Compiler for Parallel Computing*, Cambridge Mass, 1991. The MIT Press.

[29] J. W. Liu. Computational models and task scheduling for parallel sparse cholesky factorization. *Parallel Computing*, 3:327–342, 1986.

[30] 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, to appear*, Santa Clara, CA, August 1991.

[31] Parti runtime primitives for compilers, in progress. Interim report, ICASE, 1991.

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

[33] 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 , St. Malo France, pages 140–152, July 1988.

[34] M. J. Quinn and P. J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, pages 69–76, September 1990.

[35] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation.* ACM SIGPLAN, June 1989.

[36] M. Rosing and R. Schnabel. An overview of Dino - a new language for numerical computation on distributed memory multiprocessors. Technical Report CU-CS-385-88, University of Colorado, Boulder, 1988.

[37] M. Rosing, R.W. Schnabel, and R.P. Weaver. Expressing complex parallel algorithms in Dino. In *Proceedings of the 4th Conference on Hypercubes, Conurrent Computers and Applications*, pages 553–560, 1989.

[38] Y. Saad. Sparsekit: a basic tool kit for sparse matrix computations. Report 90-20, RIACS, 1990.

[39] J. Saltz, H. Berryman, and J. Wu. Runtime compilation for multiprocessors, to appear: Concurrency, Practice and Experience, 1991. Report 90-59, ICASE, 1990.

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

[41] J. Saltz, R. Das, R. Ponnusamy, D. Mavriplis, H Berryman, and J. Wu. Parti procedures for realistic loops. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, Oregon, April-May 1991.

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

[43] P. S. Tseng. *A Parallelizing Compiler for Distributed Memory Parallel Computers.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, May 1989.

[44] M. Weiser. Program slicing. *IEEE Trans. on Software Eng.*, SE-10(4):352–357, July 1984.

[45] R. D. Williams and R. Glowinski. Distributed irregular finite elements. Technical Report C3P 715, Caltech Concurrent Computation Program, February 1989.

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

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