1992

PARTI Primitives for Unstructured and Block Structured Problems

Alan Sussman
NASA, Langley Research Center, ICASE

Joel Saltz
NASA Langley Research Center, ICASE

Raja Das
NASA Langley Research Center, ICASE

S. Gupta
NASA Langley Research Center, ICASE

Dimitri Mavriplis
NASA Langley Research Center, ICASE

See next page for additional authors

PARTI Primitives for Unstructured and Block Structured Problems[1]

Alan Sussman[a], Joel Saltz[a], Raja Das[a], S Gupta[a], Dimitri Mavriplis[a] and Ravi Ponnusamy[b]

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

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

Abstract

This paper describes a set of primitives (PARTI) developed to efficiently execute unstructured and block structured problems on distributed memory parallel machines. We present experimental data from a 3-D unstructured Euler solver run on the Intel Touchstone Delta to demonstrate the usefulness of our methods.

1 Introduction

We consider tools that can be used to port irregular problems to distributed memory architectures. We specifically consider irregular problems that can be divided into a sequence of concurrent computational phases. In irregular problems, such as solving PDEs on unstructured or multiblock meshes (grids), the communication pattern depends on the input data. This typically arises due to some level of indirection in the code. We address cases in which data access patterns within each computationally intensive loop can be determined before the program enters the loop. In some problems, data access patterns are specified by integer indirection arrays. Examples of problems with these characteristics include unstructured mesh explicit and multigrid solvers, along with many sparse iterative linear systems solvers. We call this class of problems *static single-phase or multi-phase computations* (SSMPs). In other cases, programs can exhibit highly uniform local computational structure. For such problems, non-uniformities in computational patterns occur in the interfaces between regular subdomains. Examples include multiblock Navier Stokes solvers and structured adaptive multigrid problems. We will call this class of problems *irregularly coupled regular mesh* computations (ICRMs). In a different paper in this volume, a more detailed taxonomy of irregular problems is presented [11].

In the kinds of algorithms we consider here, data produced or input during a program's initialization phase play a large role in determining the nature of the subsequent

[1]This work was 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.

computation. 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.

To effectively exploit many multiprocessor architectures, we may have to carry out runtime preprocessing. This preprocessing is referred to as *runtime compilation* [36]. The purpose of runtime compilation is not to determine which computations are to be performed but instead to determine how a multiprocessor machine will schedule the algorithm's work, how to map the data structures and how data movement within the multiprocessor is to be scheduled.

In distributed memory MIMD architectures, there is typically a non-trivial communications 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 to receive.

Only recently have methods been developed to integrate the kinds of runtime optimizations mentioned above into compilers and programming environments [36]. The lack of compile-time information is dealt with by transforming the original parallel loop into two constructs called an inspector and executor [32]. During program execution, the inspector 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 closely related suites of primitives that can be used directly by programmers to generate inspector/executor pairs for SSMP and ICRM problems. These primitives carry out preprocessing that makes 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 for SSMP computations make use of hash tables [20] to allow us to recognize and exploit a number of situations in which a single off-processor datum is used several times. In such situations, the primitives only fetch a single copy of each unique off-processor distributed array reference.

In many ICRM problems there are at most a few dozen meshes (blocks) of varying sizes. If that is the case, it may be necessary to assign at least some of the meshes to multiple processors to use all of the processors available in the distributed memory parallel machine. We must consequently be prepared to deal with multiple levels of parallelism in ICRM codes. Typically ICRM applications have two levels of parallelism available. Coarse-grained parallelism is available for processing the meshes concurrently. Each mesh is a self-contained computation region that can, except for boundary conditions, be operated upon independently of the other meshes. In addition, the computation for individual blocks has fine-grain parallelism available. Applying coarse-grained parallelism will help to keep communication overhead to a manageable fraction of the computation time. However, since the number of meshes is relatively small, particularly when compared to the number of processing elements in current distributed-memory multicomputers, the coarse-grained parallelism between meshes will not provide sufficient parallel activity to keep all processors busy. The fine-grained parallelism within each block must be used to fill this gap.

Primitives for ICRM problems make it possible for programmers to embed each mesh into a subset of the processors in the distributed memory parallel machine. The primitives schedule and carry out required patterns of data movement within and between meshes.

The suite of primitives used for SSMP problems is called *PARTI* (*Parallel Automated Runtime Toolkit at ICASE*), while the suite of primitives used for ICRM problems is called *multiblock PARTI*.

Section 2 gives an overview of the PARTI routines for SSMP problems, and Section 3 provides a more detailed description of how the routines work. Section 4 discusses the multiblock PARTI routines, including a description of how to apply them to a multiblock computational fluid dynamics application. Some experimental results for using the PARTI primitives are given in Section 5. Section 6 describes other research related to supporting irregular computations, and Section 7 concludes.

2 PARTI

In this section, we give an overview of the principles and functionality of the PARTI primitives. In Section 3 we give a more detailed description of some of the more sophisticated PARTI procedures.

2.1 Parti Overview

The PARTI primitives (Parallel Automated Runtime Toolkit at ICASE) are designed to ease the implementation of computational problems on parallel architecture machines by relieving the user of low-level machine specific issues. The PARTI primitives enable the distribution and retrieval of globally indexed but irregularly distributed data sets over the numerous local processor memories. In distributed memory machines, large data arrays need to be partitioned among the 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. A processor that needs to read an array element must fetch a copy of that element from the memory of the processor in which that array element is stored. Alternately, a processor may need to store a value into an off-processor distributed array element. Thus, each element in a distributed array is assigned to a particular processor, and in order to access a given element of the array we must know the processor on which it resides, and its local address in that processor's memory. To store this information, we 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 also must be distributed among the local memories of the processors. For a P processor machine, this is accomplished by putting the first N/P elements on the first processor, the second N/P elements on the second processor, etc. Thus, 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 processor $m/P + 1$. Alternatively, we could renumber all the vertices of the unstructured grid to obtain a regular partitioning of arrays over the processors. However, our approach can easily deal with arbitrary partitions, and should enable a straightforward implementation of dynamically varying partitions, which may be encountered in the context of adaptive meshes. One primitive handles initialization

of distributed translation tables, and another primitive is used to access the distributed translation tables.

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 the locations of the data each processor needs to send and receive. In irregular problems, such as solving PDEs on unstructured meshes and sparse matrix algorithms, the communications pattern depends on the input data. In this case, it is not possible to predict at compile time what data must be prefetched. This lack of information is dealt with by transforming the original parallel loop into two constructs called an inspector and executor. During program execution, the inspector 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. The PARTI primitives can be used directly by programmers to generate inspector/executor pairs. Each inspector produces a communications *schedule*, which is essentially a pattern of communication for gathering or scattering data. In order to avoid duplicate data accesses, a list of off-processor data references is stored locally (for each processor) in a hash table. For each new off-processor data reference required, a search through the hash table is performed in order to determine if this reference has already been accessed. If the reference has not previously been accessed, it is stored in the hash table, otherwise it is discarded. The primitives thus only fetch a single copy of each unique off-processor distributed array reference.

The executor contains embedded PARTI primitives to gather or scatter data. The primitives are designed to minimize the effect on the source code, such that the final parallel code remains as close in form as possible to the original sequential code. The primitives issue instructions to gather, scatter or accumulate (i.e. scatter followed by add) data according to a specified schedule. Latency or start-up cost is reduced by packing various small messages with the same destination into one large message.

Significant work has gone into optimizing the gather, scatter and accumulation communication routines for the Intel Touchstone Delta machine. During the course of developing the PARTI primitives (originally for the Intel iPSC/860 hypercube), we experimented with many of ways of writing the kernels of our communication routines. It is not the purpose of this paper to describe these low level optimizations or their effects in detail; we will just summarize the best communication mechanism we have found. In the experimental study reported in this paper we use the optimized version of the communication routine kernels.

The communication is done using Intel forced message types. We use non-blocking receive calls (Intel irecv), and each processor posts all receive calls before it sends any data. Synchronization messages are employed to make sure that an appropriate receive has been posted before the relevant message is sent.

Communications contention is also reduced. We use a heuristic developed by Venkatakrishnan [42] to determine the order in which each processor sends out its messages. The motivation for this heuristic is to reduce contention by dividing the communication into groups of messages such that, within each group, each processor sends and receives at most one message. As Venkatakrishnan notes, this heuristic makes the tacit assumption that all messages are of equal length and in any event does not attempt to eliminate link

```
      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 1: Sequential Code

contention.

3 A Detailed View of PARTI

3.1 Primitives for Communications Scheduling

This section describes in some detail the primitives that schedule and perform movement of data between processors. 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 1 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 1 (statements S1 through S4), so the values of **x** are not modified by these loops. In Figure 1, array **y** is both read and written. These references all involve accumulations in which computed quantities are added to specified elements of **y** (statements S1 through S4).

3.2 PARTI Executor

Figure 2 depicts the *executor* code with embedded calls to Fortran PARTI procedures *dfmgather*, *dfscatter_add* and *dfscatter_addnc*. Before this code is executed, we must carry out a preprocessing phase, which is described in Section 3.3. 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 [38].

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 2 is identical to that of the loops in Figure 1. In Figure 2, we again use arrays named `x` and `y`; in Figure 2, `x` and `y` now represent arrays defined on a single processor of a distributed memory multiprocessor. On each processor, arrays `x` and `y` are declared to be larger than would be needed to store the number of array elements for which that processor is responsible. Copies of the off-processor data are placed in a buffer area beginning with `x(n_on_proc+1)`.

The PARTI subroutine calls depicted in Figure 2 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 2, off-processor data is obtained from array `x` defined on each processor.

The PARTI procedures *dfscatter_add* and *dfscatter_addnc*, in statements S2 and S3 Figure 2, 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 indexes 1 and `n_on_proc`. The distinctions between *dfscatter_add* and *dfscatter_addnc* will be described in Section 3.4.

In Figure 2, several data items may be accumulated to a given off-processor location in loop L1 or in loop L2.

3.3 PARTI Inspector

In this section, we outline how to perform the preprocessing needed to generate the arguments required by the code in Figure 2. This preprocessing is depicted in Figure 3.

The way in which the nodes of an irregular mesh are numbered frequently does not have a useful correspondence to the connectivity pattern of the mesh. When we

```
      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 2: 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 3: Inspector Code for Each Processor

partition such a mesh in a way that minimizes interprocessor communication, we may need to assign arbitrary mesh points to each processor. The PARTI procedure *ifbuild_translation_table* (S1 in Figure 3) allows us to map a globally indexed distributed 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 3). If a given processor needs to obtain a data item 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 item 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 2. We will first describe *flocalize* (S2 in Figure 3). 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, (part_edge_list in S2), and
3. the 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 (local_edge_list) that is used to specify the access pattern of arrays **x** and **y** in S1 and S2 of Figure 2,
3. and the number of distinct off-processor references found in edge_list (n_off_processor in S2).

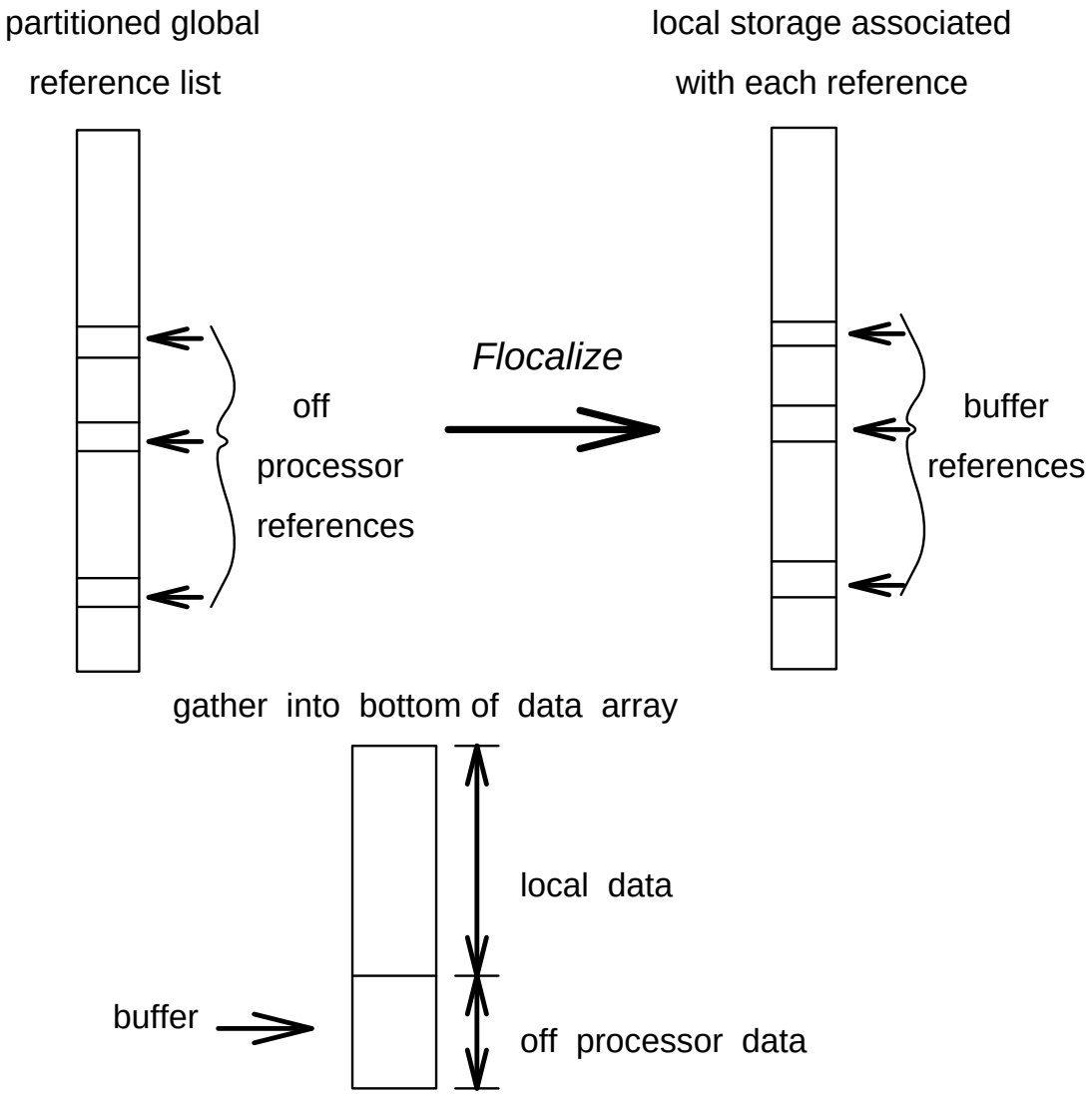

Figure 4: Flocalize Mechanism

A sketch of how the procedure *flocalize* works is shown in Figure 4. The array **edge_list** shown in Figure 1 is partitioned between processors. The **part_edge_list** passed to *flocalize* on each processor in Figure 3 is a *subset of* **edge_list** depicted in Figure 1. We cannot use **part_edge_list** to index an array on a processor since **part_edge_list** refers to globally indexed elements of arrays **x** and **y**. *Flocalize* modifies 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)**. When *flocalize* produced **local_edge_list** from **part_edge_list**, the off-processor references were 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 1). In Figure 1, no assignments to **x** are carried out. At the beginning of the program in Figure 2, 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 3) removes 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 2 obtains off-processor data using *two* schedules; *edge_sched* produced by *flocalize* (S2 Figure 3) and *incremental_face_sched* produced by *fmlocalize* (S4 Figure 3).

The pictorial representation of the *incremental schedule* is given in Figure 5. 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 5. 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 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 an 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 (part_face_list in S4),
3. the number of globally indexed distributed array references (4*n_face in S4),
4. the number of pre-existing schedules that need to be examined 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),

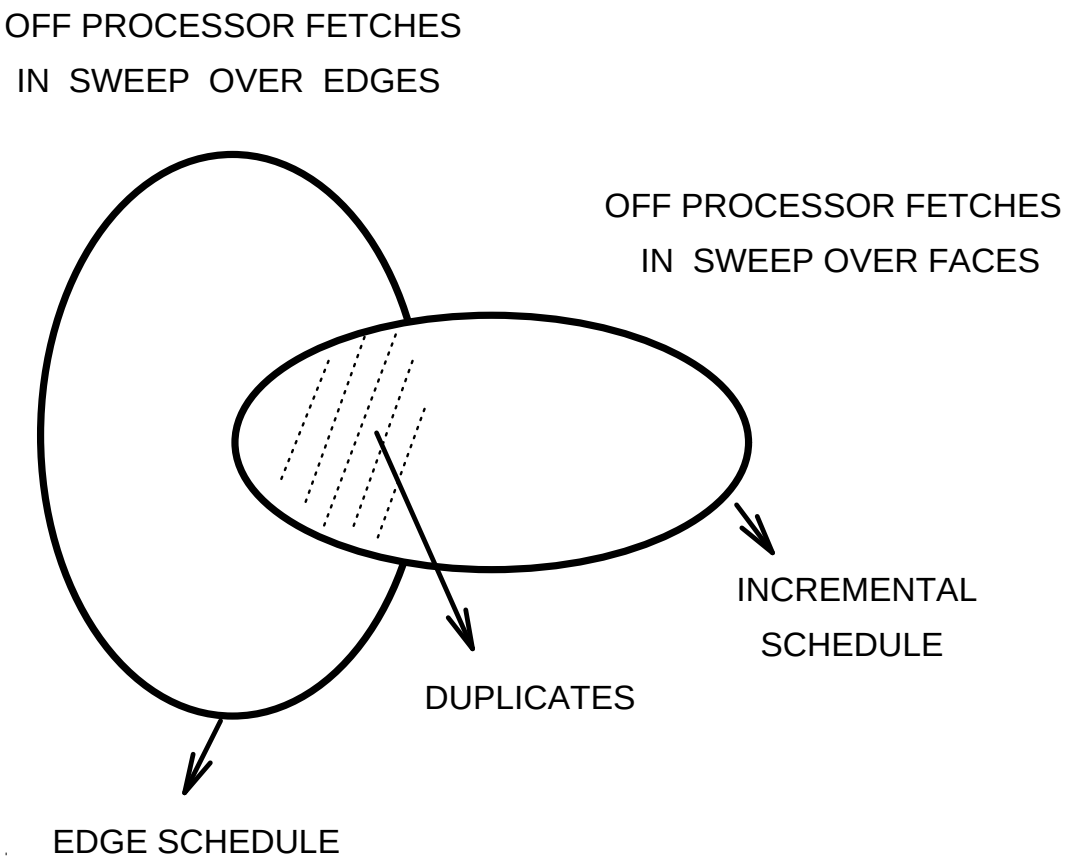

Figure 5: Incremental schedule

2. an *incremental schedule* that includes only off-processor data accesses not included in the pre-existing schedules (incremental_face_sched in S4),

3. an integer array (local_face_list in S4) that is used to specify the access pattern of arrays **x** and **y** in statements S3 and S4 of the executor code (Figure 2),

4. the number of distinct off-processor references in face_list (n_off_proc_face in S4),

5. the number of distinct off-processor references not encountered in any other schedule (n_new_off_proc_face in S4),

6. and a buffer_mapping - to be discussed in Section 3.4.

3.4 A Return to the Executor

We have already discussed *dfmgather* in Section 3.2 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. In this situation,the 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 2). 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 2, 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 statement S3 from Figure 2 (*dfscatter_addnc* stands for *dfscatter_add* non-contiguous).

3.5 Automatic Inspector/Executor Generation

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. Joint work with groups at Rice and Syracuse is underway to employ PARTI as the runtime support for a compiler that automatically generates distributed memory programs that make effective use of incremental and non-incremental schedules. This compiler is based on the Parascope parallel programming environment [22] and compiles Fortran D [21]. Another group, at the University of Vienna, has already employed PARTI for the runtime support in their distributed memory compiler [7].

4 Multiblock Parti

We are developing methods for parallelizing programs with irregularly coupled regular meshes (ICRMs), commonly known as multiblock applications, to distributed memory parallel computers. In order to ensure that our techniques are applicable to real-world problems, we have begun our research with a specific multiblock problem from the domain of computational fluid dynamics.

In many problems there are at most a few dozen blocks of varying sizes. We can assume that we will have to assign at least some of the blocks to multiple processors, we must consequently be prepared to deal with multiple levels of parallelism in ICRM codes. Typically ICRM applications have two levels of parallelism available. Coarse-grained parallelism is available for processing the blocks concurrently. Each block is a self-contained computation region that can, except for boundary conditions, be operated upon independently of the other blocks. In addition, the computation for individual blocks has fine-grain parallelism available. Applying coarse-grained parallelism will help to keep communication overhead to a manageable fraction of the computation time.

4.1 Problem Overview

The application we are investigating is a problem from the domain of computational fluid dynamics. The serial code was developed by V. Vasta, M. Sanetrik and E. Parlette at the NASA Langley Research Center [41], and solves the thin-layer Navier-Stokes equations for a fluid flow over a three-dimensional surface with complex geometry. The problem geometry is decomposed into between one and a few dozen distinct blocks, each of which is modeled with a regular, three-dimensional, rectangular grid. An example of the multiblock grid structure surrounding an airplane (an F-18) is shown in Figure 6. The meshes are shown intersecting the solid surface of the airplane, and the various colors correspond to different blocks.

The boundary conditions of each block are enforced by simulating any of several situations, such as viscous and inviscid walls, symmetry planes, extrapolation conditions, and interaction with an adjacent block. The size of each block, its boundary conditions and adjacency information are loaded into the program at run-time. For this application, the same program is run on all blocks. However, different subroutines will be executed when applying the boundary conditions on different blocks. In general, the code used to process each block of an ICRM application may be different.

The sequence of activity for this program is as follows:

Read block sizes, boundary conditions and simulation parameters,

Repeat (typically large number of times):

A. Apply boundary conditions to all blocks,

B. Carry out computations on each block.

The main body of the program consists of an outer sequential loop, and two inner parallel loops. Each of the inner loops iterates over the blocks of the problem, the first applying boundary conditions (Step A), which may involve interaction with other blocks, and the second loop advancing the physical simulation one time step in each block

Figure 6: Multiblock grid intersecting the surface of an F-18

(Step B). Partitioning of the parallel loops is the source of the coarse-grained parallelism for the application. Furthermore, within each iteration of the loop that implements Step B there is fine-grained parallelism available in the form of (large) parallel loops.

4.2 The Multiblock PARTI Library

Several forms of run-time support are required for ICRM applications. First, there must be a means for expressing data layout and organization on the processors of the distributed memory parallel machine. Second, there must be methods for specifying the movement of data required both because of partitioning of individual meshes (intra-block parallelism) and because of interactions between different meshes (inter-block parallelism). Third, there must be some way of transforming distributed array indexes specified in global coordinates (as in the sequential code) into local indexes on a given processor in the distributed memory parallel machine.

Integration of the required functionality into the Fortran D language [16] is currently underway. As a preliminary step, we have defined a library of subroutines for expressing this functionality in Fortran programs, and are using them to test our support for ICRMs. The data layout support provided by the library corresponds to Fortran D style declarations of distributed arrays. The run-time activities that directly handle data communication are generated from the data usage patterns in the user program (either by the user or eventually by a compiler), and consist of subroutine calls to:

1. build schedules (communication patterns, as described in Section 2) for either intra-block or inter-block communication,

2. perform data movement using a previously built schedule,

3. and transform a global distributed array index into a local array index.

One major difference between PARTI and multiblock PARTI is that building schedules for ICRM codes does not require interprocessor communication, since each processor knows the layout of all the distributed arrays. Therefore no distributed translation table is required. Similarly, in multiblock PARTI, transforming a global distributed array index into a local index does not require a lookup into a (distributed) translation table, but only requires computing the proper local index using the (local) data structure associated with each distributed array. We now discuss the run-time support routines in more detail.

4.2.1 Data Layout

The binding of blocks to processors has important performance implications. Load balance plays a crucial role in determining computational efficiency. Since the amount of computation associated with each block is directly proportional to the number of elements in the block, good load balancing is achieved by binding processors to blocks in a ratio proportional to their sizes. In our implementation, this mapping is under user control.

The principal abstraction for dealing with data placement is the *decomposition*. However, unlike Fortran D, where decompositions are bound to the entire processor set, we map decompositions to subsets of the processors. The mechanism for specifying this arrangement is a subroutine called *embed*. *Embed* binds a decomposition to a rectangular subregion of another decomposition. Any number of decompositions may be embedded into a single root decomposition. The root decomposition is mapped onto the entire set of physical processors. Embedded decompositions are mapped onto subsets of these processors based on the relative size and location of the subregion in the root decomposition to which they are bound. This methodology can easily be extended recursively to support an arbitrary sequence of embeddings, although for most ICRM applications we are aware of a two level decomposition hierarchy appears to be sufficient.

For the Navier-Stokes application, we use a one-dimensional decomposition for the root level, and embed 3-dimensional blocks into it. For example, if two blocks, one of size $10 \times 10 \times 10$ and the other $5 \times 5 \times 10$ were to be mapped onto the physical processing resource, a root-level decomposition of size 1250 would be used. The first block would be embedded into locations 1 through 1000 of this decomposition, and the second block into locations 1001 through 1250. This implies that 4/5 of the processors are used to compute for the first block, and 1/5 of the processors are used for the second block.

The *distribute* subroutine defines the type of distribution for each dimension of a decomposition. *Distribute* supports three types of distributions for the N elements of one dimension of a decomposition, to be partitioned onto P processors (assuming that both decomposition elements and processors are numbered starting at 1):

1. block, in which the first N/P elements are assigned to the first processor, the second N/P to the second processor, etc.,

2. cyclic, in which processor i is assigned all elements with index j such that $i = j \bmod P$,

3. and undistributed.

While a decomposition is an abstract specification of a problem domain, another subroutine is required to map a particular distributed array with respect to a decomposition. The *align* subroutine conforms a distributed array with a decomposition, in addition allowing the specification of rotation (so that any array dimension can be aligned with any decomposition dimension) and of *ghost cells* for each dimension. These ghost cells will contain copies of distributed array elements residing on other processors that are required to perform local computation (caused by partitioning a single block to obtain fine-grained parallelism). The use of decompositions as an abstraction of a problem domain allows multiple distributed arrays to be mapped in exactly the same way, even if two arrays are not exactly the same size (e.g. the size of one is some multiple of the size of the other, as in a multigrid application), or have dimensions that are rotated with respect to each other (e.g. matrices aligned so that the rows of one matrix are mapped in the same way as the columns of another matrix). Another possibility is to align only some of the dimensions of a distributed array to an entire decomposition (e.g. align a 4-D array with a 3-D decomposition). In that case, all the elements in the unaligned dimensions of the distributed array are allocated on all processors that contain decomposition elements.

4.2.2 Interprocessor Communication

Two types of communication are required in ICRM applications: intra-block communication because a single block may be partitioned across the processors of the distributed memory parallel machine, and inter-block communication because of boundary conditions between blocks, caused by the assignment of blocks to different processors to obtain coarse-grained parallelism. As for the PARTI primitives for unstructured mesh computations, communication is performed in two phases. First, a subroutine is called to build a communication *schedule* that describes the required data motion, and then another subroutine is called to perform the data motion (sends and receives on a distributed memory parallel machine) using a previously built schedule. Such an arrangement allows a schedule to be used multiple times in an iterative algorithm (such as the Navier-Stokes multiblock algorithm), so long as the data layout does not change. This amortizes the cost of building schedules, so that the preprocessing time should not be a significant part of the execution time of this type of program.

The communication primitives include a procedure *exch_sched*, which computes a schedule that is used to direct the filling of overlap cells along a given dimension of a distributed array. *Exch_sched* executes on each processor that contains a part of the distributed array, and, for a given processor i, determines both which other processors require data that is stored on processor i, and which other processors store data that processor i requires.

The primitive *subarray_sched* carries out the preprocessing required to copy the contents of a regular section [19], *source*, in one block into a regular section, *destination*, in another (or the same) block. The interactions between blocks for ICRM applications are limited to the exchange of regular sections. The *subarray_sched* primitive supports data moves between arbitrary rectangular portions of two blocks, and can transpose the data along any dimension. *Subarray_sched* produces a schedule which specifies a pattern of intra-processor data transfers (for the parts of the source and destination subsections

that reside on the same processor), along with a set of send and receive calls for inter-processor communication. On a given processor, i, *subarray_sched* determines whether it owns any portion of *source*. If i does own some portion, *source_i*, of *source*, *subarray_sched* computes the processors to which various parts of *source_i* must be sent. Similarly, *subarray_sched* also computes whether processor i owns any portion of *destination* and, if so, determines which other processors send messages to processor i.

The schedules produced by *exch_sched* and *subarray_sched* are employed by a primitive called *data_move* that carries out both interprocessor communication and intra-processor data copying.

4.2.3 Distributed Array Index Transformation

The final form of support provided by the library for ICRMs is to transform all indexes into distributed arrays from the global value (an index into the whole distributed array) to a local index on the processor executing a distributed array reference. For a loop that only uses the loop index to reference into one distributed array (or multiple distributed arrays mapped identically), the index transformation can be performed in the loop header, only modifying the loop bounds to iterate over the indexes of the local distributed array elements. Two primitives, *local_lower_bound* and *local_upper_bound*, are provided for transforming loop bounds (returning, respectively, the lower and upper local indexes of a given dimension of the referenced distributed array). In general, however, each distributed array reference (read or write) must have the array index transformed from a global to a local reference for correct parallel execution. Techniques for collecting all the references to multiple distributed arrays in a single loop and properly transforming indexes are complex, and have been investigated by other researchers [21].

4.3 An Example

An example of the structure of a parallelized explicit multiblock code should help clarify the use of the library routines. We will display both the parts of the code that declare the distributed arrays and the parts that build and use schedules for intra-block and inter-block communication. Multigrid code would have the same general structure, with loops over the grid levels surrounding the code for the explicit time step. Multigrid code also requires transferring data between multigrid levels, which can be done using the *subarray_exch* primitive. The pseudo-code is shown in Figure 7. For simplicity, assume that we already know the global sizes of all the blocks in the data array x.

The declarations of the distributed arrays are fairly straightforward. The various blocks will all be stored in one array x, and a separate pointer array will contain the starting positions of each block. The decomposition $D1$ is mapped onto the entire set of physical processors that the program runs on, while each decomposition in $D3$ is embedded into a part of the physical processor set (physical processors are assigned based on the relative sizes of the various blocks). Each block in x is then aligned with its corresponding decomposition (in this example each decomposition is used for only one distributed array).

In this example, the distribution of the distributed array x does not change, so schedules for data movement may be computed once, and saved for multiple later uses. Therefore, in the main loop body only calls to the *data_move* subroutine are required,

1. Allocate a 3-D data array x, large enough for all the block portions to be stored locally (including ghost cells).

2. Create a 1-D **decomposition**, $D1$, with size equal to the total number of elements in x (the sum of the sizes of all the blocks, without ghost cells).

3. Create an array of 3-D **decompositions**, $D3[num\_blocks]$. Each element of $D3$ corresponds to one block, and each decomposition is the same size in every dimension as its corresponding block.

4. **Embed** decomposition $D3[1]$ into $D1$ at position 1, and all other decompositions $D3[i]$ into $D1$ after $D3[i-1]$ (i.e. $D3[i]$ starts right after $D3[i-1]$ ends).

5. **Distribute** each decomposition in $D3$ (e.g. block-wise in each of its dimensions).

6. **Align** each block in x with its corresponding decomposition in $D3$ (i.e. align block i with $D3[i]$). Also specify the number of ghost cells required in each dimension.

7. Fill in pointer array $blocks\_x$, so that $blocks\_x$ contains the indexes for the start of each individual block in x. This can be done now that the local sizes of all the blocks can be determined from the declarations for the distributed array (including ghost cells).

8. Build and save schedules for all interfaces between blocks, using **subarray_exch**.

9. Build and save schedules for filling in ghost cells of each block, using **exch_sched**.

10. For each time step do:

    (a) Update boundary conditions - for each block interface, call **data_move** with the corresponding previously built schedule (from **subarray_exch**).

    (b) For each block in x do:

        i. Fill in ghost cells, with a call to **data_move**, using a previously built schedule for the block (from **exch_sched**).

        ii. For each locally owned element of the block, perform the local computation - the loop bounds for this iteration are obtained from **local_lower_bnd** and **local_upper_bnd** applied to the current block.

Figure 7: Parallel multiblock code for each processor, using multiblock PARTI

both for inter-block and intra-block communication. Global to local index translation is performed on the innermost loops that iterate over the local elements of the distributed data array x, using the loop bound adjustment subroutines. This assumes that the innermost loop indexes are only used to index into distributed array x, and not for other purposes.

5 Experimental Results for an Unstructured Mesh

We summarize the results of some of the experiments we have carried out to evaluate the performance impact of our optimizations. These experiments were carried out on the Intel Touchstone Delta machine. For purposes of comparison, we cite performance numbers obtained from an optimized Cray YMP version of this code [31]. A more detailed account of this experimental work may be found in [13].

The test case we report here involves the computation of a highly resolved flow over a three-dimensional aircraft configuration. We employed both an explicit algorithm and a V cycle multigrid algorithm. The mesh employed for the explicit algorithm, which corresponds to the finest mesh employed in the multigrid calculation, contains 804,056 points and approximately 4.5 million tetrahedra. We believe this is the largest unstructured grid Euler solution attempted to date. In Figure 8, we depict the second mesh used in the multigrid sequence (we do not show the 804K mesh due to printing and resolution limitations). The mesh shown has 106,064 points and 575,986 tetrahedra. For this case, the freestream Mach number is 0.768 and the incidence is 1.16 degrees. The computed Mach contours are also shown in Figure 8, where good resolution of the shock on the wing is observed.

We employed the recursive spectral partitioning algorithm to carry out partitioning [33, 39]. Williams [43] compared this algorithm with binary dissection [5] and simulated annealing methods for partitioning two-dimensional unstructured mesh calculations. He found that recursive spectral partitioning produced better partitions than binary dissection. Simulated annealing in some cases produced better partitions but the overhead for simulated annealing proved to be prohibitive even for the relatively small meshes employed (the largest had 5772 elements). Venkatakrishnan [42] and Simon [39] also reported favorable results with the spectral partitioner. We carried out preliminary performance comparisons between binary dissection and the recursive spectral partitioning and found that recursive spectral partitioning gave superior results on an iPSC/860 hypercube on our three dimensional meshes. The results we report all have been obtained using recursive spectral partitioning to partition all meshes. Partitioning was performed on a sequential machine as a preprocessing operation. We use the optimized version of the communications kernels which employ forced message types, non-blocking receives (irecv), and employ Venkatakrishnan's heuristic to determine the order in which messages are sent.

The single mesh algorithm achieved a rate of 778 Mflops on 256 processors of the Delta machine, and 1496 Mflops on the full 512 processor configuration of the Delta. The V cycle multigrid algorithm achieved a rate of 1200 Mflops on 512 processors. We implemented the explicit Euler solver with and without incremental scheduling optimization. In Table 1, we depict:

computational rate in Mflops,

Figure 8: Coarse Unstructured Mesh and Mach Contours about an Aircraft Configuration with Single Nacelle

Method	Time/ Iteration (seconds)	Mflops	Preprocessing Time seconds
No Incremental Scheduling	4.18	947	2.73
Incremental Scheduling	2.65	1496	2.99

Table 1: Explicit Unstructured Euler Solver on 804K Mesh on 512 Delta Processors-Incremental v.s. Non-Incremental Scheduling

the time required per iteration, and

the preprocessing time needed to generate all communication schedules.

We note that incremental scheduling leads to a roughly 35% reduction in total time per iteration in this problem. The preprocessing time increases only modestly when we use incremental scheduling and is roughly equal to the cost of a single parallelized iteration.

The same problem was run on the CRAY YMP-8 machine, using all eight processors in dedicated mode. The CRAY autotasking software was used to parallelize the code for this architecture. Both the single grid and multigrid codes achieved a computational rate of 750 Mflops on all eight processors, which corresponds to a speedup of roughly 7.5 over the single processor performance.

6 Related Research

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, 4, 15, 28, 44].

Several researchers have developed programming environments that are targeted towards particular classes of irregular or adaptive problems. Williams [44] describes a programming environment (DIME) for calculations with unstructured triangular meshes using distributed memory machines. Baden [3] has developed a programming environment targeted towards particle computations. This programming environment provides facilities that support dynamic load balancing. DecTool [12] 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 [1, 9, 10, 14, 17, 18, 23, 25, 26, 27, 34, 35, 36, 40, 45]. Runtime compilation methods

are employed in four of these projects; the Fortran D project [21], the Kali project [23], Marina Chen's work at Yale [30] and our PARTI project [32, 36, 37]. The Kali compiler was the first compiler to implement inspector/executor type runtime preprocessing [23] and the ARF compiler was the first compiler to support irregularly distributed arrays [36]. 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].

Initial efforts toward runtime and compiler support for block structured problems within the PARTI project are described in [6, 8]. Work has also been done at GMD in Germany to parallelize block structured grid algorithms [29], and to provide software support for such efforts [24].

7 Conclusions

We have discussed tools that can be used to port irregular problems to distributed memory parallel machines. We have described PARTI primitives to support irregular problems on both unstructured and multiblock structured meshes. As the experimental results of using the PARTI primitives to parallelize an unstructured grid Euler solution in Section 5 show, our methods can be used to efficiently execute irregular problems on highly parallel distributed memory machines. In the future, we should obtain similar, or better, efficiency using the multiblock PARTI primitives for the multiblock CFD application described in Section 4.1. Multiblock codes should obtain better performance from each processor in the distributed memory parallel machine than unstructured codes, because of more regular access to local memory. Also, the multiblock primitives do not require interprocessor communication to build schedules (as do the PARTI primitives for unstructured problems). Further work is continuing to expand the class of irregular problems that are supported by the PARTI primitives, and at the same time we are continuing to improve the performance of the existing implementations.

Acknowledgments

We would like to thank Horst Simon for providing us with his recursive spectral partitioner and Rob Vermeland and CRAY Research Inc. for providing dedicated time on the CRAY YMP-8 machine. This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. We gratefully acknowledge NASA Langley Research Center for providing access to this facility.

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. *SIAM J. Sci. and Stat. Computation.*, 12(1), January 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] 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.

[7] P. Brezany, M. Gerndt, V. Sipkova, and H.P. Zima. SUPERB support for irregular scientific computations. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-92)*, pages 314–321. IEEE Computer Society Press, April 1992.

[8] Craig Chase, Kay Crowley, Joel Saltz, and Anthony Reeves. Parallelization of irregularly coupled regular meshes. Technical Report 92-1, ICASE, NASA Langley Research Center, January 1992.

[9] M. C. Chen. A parallel language and its compilation to multiprocessor architectures or VLSI. In *2nd ACM Symposium on Principles of Programming Languages*, January 1986.

[10] 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.

[11] Alok Choudhary, Geoffrey Fox, Sanjay Ranka, Seema Hiranandani, Ken Kennedy, Charles Koelbel, and Joel Saltz. Software support for irregular and loosely synchronous problems. In *Proceedings of the Symposium on High-Performance Computing for Flight Vehicles*, December 1992.

[12] 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.

[13] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives, AIAA-92-0562. In *Proceedings of the 30th Aerospace Sciences Meeting*, January 1992.

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

[15] 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.

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

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

[18] 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 and Practice of Parallel Programming*, pages 73–82, April 1991.

[19] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[20] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman. Performance of hashed cache data migration schemes on multicomputers. *Journal of Parallel and Distributed Computing*, 12:415–422, August 1991.

[21] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, pages 139–176. Elsevier Science Publishers B.V., 1992.

[22] K. Kennedy, K.S. McKinley, and C.-W. Tseng. Interactive parallel programming using the Parascope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.

[23] 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.

[24] Max Lemke and Daniel Quinlan. P++, a C++ virtual shared grids based programming environment for architecture-independent development of structured grid applications. Technical Report 611, GMD, February 1992.

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

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

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

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

[29] Guy Lonsdale and Anton Schuller. Parallel and vector aspects of a multigrid Navier-Stokes solver. Technical Report 550, GMD, June 1991.

[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*, Santa Clara, CA, August 1991.

[31] D. J. Mavriplis. Three dimensional multigrid for the Euler equations. *AIAA paper 91-1549CP*, pages 824–831, June 1991.

[32] 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.

[33] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Mat. Anal. Appl.*, 11:430–452, 1990.

[34] Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80. ACM Press, June 1989.

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

[36] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency: Practice and Experience*, 3(6):573–592, 1991.

[37] 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.

[38] 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.

[39] 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.

[40] Ping-Sheng Tseng. *A Parallelizing Compiler For Distributed Memory Parallel Computers*. PhD thesis, Carnegie Mellon University, May 1989. Also available as Technical Report CMU-CS-89-148.

[41] Veer N. Vatsa, Mark D. Sanetrik, and Edward B. Parlette. Development of a flexible and efficient multigrid based multiblock flow solver. Submitted to the 31st AIAA Aerospace Sciences Meeting, January 1993.

[42] V. Venkatakrishnan, H. D. Simon, and T. J. Barth. A MIMD implementation of a parallel Euler solver for unstructured grids, submitted to Journal of Supercomputing. Report RNR-91-024, NAS Systems Division, NASA Ames Research Center, Sept 1991.

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

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

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