NASA Contractor Report 187515
ICASE Report No. 91-13

ICASE

DISTRIBUTED MEMORY COMPILER DESIGN FOR SPARSE PROBLEMS

Janet Wu
Joel Saltz
Harry Berryman
Seema Hiranandani

Contract No. NAS1-18605
January 1991

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Distributed Memory Compiler Design for Sparse Problems [4]

Janet Wu [1]
Joel Saltz [2]
Harry Berryman [1 2]
Seema Hiranandani[3]

ABSTRACT

In this paper we describe and demonstrate a compiler and runtime support mechanism. The methods presented here are capable of solving a wide range of sparse and unstructured problems in scientific computing. The compiler takes as input a Fortran 77 program enhanced with specifications for distributing data, and the compiler outputs a message passing program that runs on a distributed memory computer. The runtime support for this compiler is a library of primitives designed to efficiently support irregular patterns of distributed array accesses and irregular distributed array partitions. We present a variety of Intel iPSC/860 performance results obtained through the use of this compiler.

[1]Computer Science Department Yale University New Haven CT 06520
[2]ICASE, NASA Langley Research Center Hampton VA 23665
[3]Computer Science Department Rice University Houston Tx 77251
[4]Research supported by the National Aeronautics and Space Administration under NASA contract NAS1-18605 while the authors were in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665, and by NSF grant ASC-8819374.

1 Introduction

During the past few years, a number of researchers have proposed integrating runtime optimization methods into compilers for distributed memory multiprocessors. These optimizations are essential in scientific codes that include sparse matrix solvers, or in programs that solve partial differential equations using adaptive and unstructured meshes. We first identified a set of relevant numerical codes that required runtime optimizations. After identifying this set, we performed extensive experimental research on these codes. The results of our experiments not only identified the major performance bottlenecks in these codes but also helped us develop a rich set of optimizations useful and essential to generating reasonably efficient code for this class of problems on distributed memory machines. Once we developed a collection of run time optimizations, we built a compiler that identifies irregular computations and performs transformations to enhance the code.

The compiler takes as input a simplified Fortran 77 program enhanced with specifications for distributing data, it outputs a message passing Fortran program for the Intel iPSC/860 parallel computer. The compiler consists of two distinct layers. The bottom layer is a library of runtime procedures (Parti - Parallel Automated Runtime Toolkit at ICASE) designed to efficiently support irregular patterns of distributed array accesses. The top layer is a compiler that carries out program transformations and embeds the Parti procedures. The Parti procedures support a variety of operations that include off processor data fetches , off processor store updates on reduction operations performed on global data structures and storage of non-local data. Parti also supports non-uniform distributed array partitions in which each distributed array element can be assigned to an arbitrary processor. A multicomputer program is generated in which all distributed memory accesses are carried out using embedded procedures.

It must be emphasized that the goal of this project is not to develop a production quality compiler, but to demonstrate that run time optimizations can be automatically and efficiently generated by a compiler. Most of the complexity of our system is in the Parti procedures. The Parti procedures have been developed so that that transformations needed to embed the appropriate primitives can be implemented with relative ease in distributed memory compilers. It may be noted that while this system's top layer is experimental and is far from being production quality code, the lower layer is currently being distributed [6].

The details of the transformations performed by the ARF compiler are

described in section 2. Section 3 describes the Parti run time primitives that have been implemented and incorporated in the compiler. In section 4 we describe the ARF language and the overall compiler strategy that demonstrates the interaction between the two layers of the ARF compiler. We describe the compiler in the context of two code examples. These examples are written in ARF and translated to iPSC/860 code by our compiler. In Section 5 we report experimental numbers for the codes compiled by the ARF compiler. In Section 6 we describe the relationship between our work and other related research projects in the area and we conclude in Section 7.

2 Distributed Memory Inspectors and Executors

In distributed memory machines, large data arrays need to be partitioned between local memories of processors. These partitioned data arrays are called *distributed arrays*. We follow the usual practice of assigning long term storage of distributed array elements to specific memory locations in the machine. Non-local reads require that a processor 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 in a non-local distributed array element requiring the processor to write to non-local memory. An issue that arises at this point is where does a processor store copies of off-processor data. Due to the irregular nature of the access pattern, it is not efficient to store the elements in temporary arrays or overlap areas proposed by Gerndt [12]. Both these storage schemes result in large wastage of memory. We store local copies of off-processor distributed array elements in hash tables called *hashed caches*. Hash tables result in less wastage of memory and quick access of off-processor data. Run time primitives are implemented to manage the hashed caches. These primitives initialize the hashed caches, store and retrieve data from them and flush the hashed caches when appropriate. During program execution, a hash table records off-processor fetches and stores. We are consequently able to recognize when more than one reference is being made to the same off-processor distributed array element, so that only one copy of that element need be fetched or stored.

In distributed memory MIMD architectures, there is typically a non-trivial communications latency or startup cost [8]. As an optimization we block messages in order to increase the message size and reduce the number of messages. This optimization can be achieved by precomputing what data each processor needs to send and receive. The preprocessing needed to per-

Each processor P:

- Preprocesses its own loop iterations
- Records off-processor fetches and stores in hashed cache
- Finds send/receive calls required for data exchange
  1. P generates list of all off-processor data to be fetched
  2. P sends messages to other processors requesting copies of required data
  3. Other processors tell P which data to send
  4. Send/Receive pairs generated and stored

Figure 1: Inspector For Parallel Loop on Distributed Memory Multiprocessor

form this optimization results in the generation of an *inspector* loop. Figure 1 describes the form of the inspector loop that is generated assuming the original loop is parallel and thus blocking messages is legal. The distribution of parallel loop indices to processors determines where computations are to be performed. We assume that all distributed arrays referenced have been defined and initialized and that loop iterations have been partitioned among processors.

During the inspector phase, we carry out a set of interprocessor communications that allows us to anticipate exactly which send and receive communication calls each processor must execute prior to executing the loop. By contrast, individual fetches and stores carried out during the actual computation would result in expensive, inefficient and awkward code [23]. For example, in such a case processor A might obtain the contents of a distributed array element which is not on A by sending a message to processor B associated with the array element. Processor B would have to be *programmed* to anticipate a request of this type, to satisfy the request and to return a responding message containing the contents of the specified array element.

The inspector loop transformation described above assumes computing the processor on which the non-local data resides is straight forward. For

example, if a one dimensional array is distributed in a block manner, simple functions can be used to compute the processor and local offset of a particular array element. However, there are many situations in which simple, easily specified distributed array partitions are inappropriate. In computations that involve an unstructured mesh, we attempt to partition the problem so that each processor performs approximately the same amount of work to achieve load balancing and to minimize communication overhead. Typically, it is not possible to express the resulting array partitions in a simple way. By allowing an arbitrary assignment of distributed array elements to processors, we have the additional burden of maintaining a data structure that describes the partitioning. The size of this data structure must be the same as the size of the the irregularly distributed array. We call this data structure a *distributed translation table.* Distributed translation tables are partitioned between processors in a simple straightforward manner described in Section 3.4.

In order to access an array element, we need to know where the element is stored in the memory of the distributed machine. This information is obtained from the distributed translation table. When a distributed translation table is used to describe array mappings, inspectors must be modified so that they access the distributed table. The modifications made to an inspector are outlined in Figure 2. In this case, the distributed translation table is used to determine the processor on which an element resides.

Once the preprocessing is completed, every processor knows exactly which non-local data elements it needs to send to and receive from the other processors. we are therefore in a position to carry out the necessary communication and computation. The loop is transformed into an *executor* loop. Figure 3 outlines the steps involved and they apply to irregular and regular array mappings. The initial data exchange phase follows the plan established by the inspector. When a processor obtains copies of non-local distributed array elements, the copies are written into the processor's hashed cache. Once the communication phase is over, each processor carries out its computation. Each processor uses locally stored portions of distributed arrays along with non-local distributed array elements stored in the hashed cache. When the computational phase is finished, distributed array elements to be stored off-processor are obtained from the hashed cache and sent to the appropriate off-processor locations. In the next section we describe the details of the Parti run time primitives that may be invoked during the inspector and executor phases.

Each processor P:

- Preprocesses its own loop iterations
- Records off-processor fetches and stores in hashed cache
- Consults distributed translation table to
  - Find location in distributed memory for each off-processor fetch or store
- Finds send/receive calls required for data exchange
  1. P generates list of all off-processor data to be fetched
  2. P sends messages to other processors requesting copies of required data
  3. Other processors tell P which data to send
  4. Send/Receive pairs generated and stored

Figure 2: Inspector For Parallel Loop Using Irregular Distributed Array Mapping

- Before loop or code segment

    1. Data to be sent off-processor read from distributed arrays
    2. Send/receive calls transport off-processor data
    3. Data written into hashed cache

- Computation carried out

    - off-processor reads/writes go to hashed cache

- At end of loop or code segment

    1. Data to be stored off-processor is read from hashed cache
    2. Send/receive calls transport off-processor data
    3. Data written back into distributed arrays for longer term storage

Figure 3: Executor For Parallel Loop on Distributed Memory Multiprocessor

3 Parti primitives

The Parti run time primitives can be divided into three categories; primitives that may be invoked during the inspector phase, executor phase or both inspector and executor phase. The *scheduler* primitive invoked during the inspector phase, determines the send and receive calls that are needed during the executor phase. These calls may be to either scatter, gather or perform reduction operations during the executor phase. The distributed translation table mentioned earlier is used during the inspector phase. The hashed cache primitives are used during the inspector and executor phases. The next section describes the details of the scheduler, distributed translation table, scatter, gather, reduction and hashed cached primitives.

3.1 The Scheduler Primitive

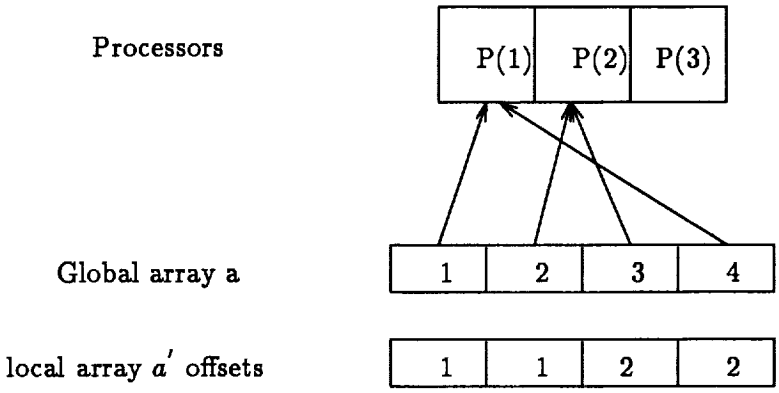

Figure 4: Mapping of a Global Array to Processors

We will use a simple example to illustrate the preprocessing carried out by the scheduler. Assume we have a distributed array a that is partitioned among three processors in an irregular fashion as depicted in Figure 4 and there is a loop computation such that the access pattern of array a is as shown in Figure 5. Each processor stores its elements of distributed array a in a *local* array a'. Thus processor P_1 needs to fetch array element $a(3)$

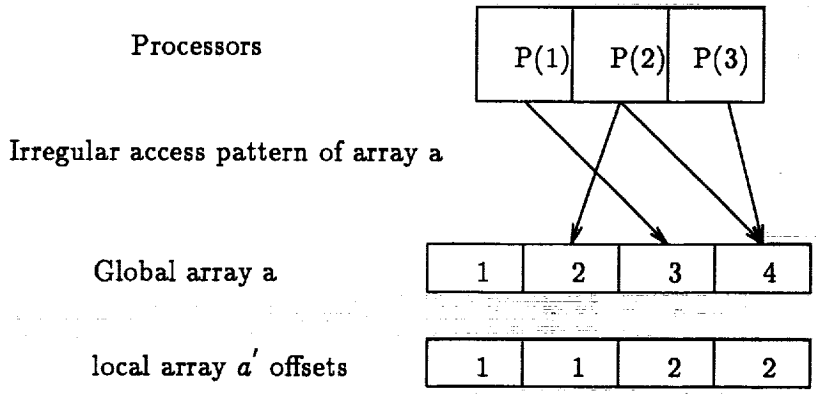

Figure 5: Irregular Access Pattern

or element $a'(2)$ of the local array from processor P_2 and processors P_2 and P_3 needs to fetch $a(4)$ or element $a'(2)$ of the local array from P_1. Recall that the task of the *scheduler* is to anticipate exactly which send and receive communications must be carried out by each processor. The scheduler first figures out *how many* messages each processor will have to send and receive during the data exchange that takes place in the executor phase. Defined on each processor $\mathtt{P}^{\mathtt{i}}$ is an array $\mathtt{nmsgs}^{\mathtt{i}}$. Each processor sets its value of $\mathtt{nmsgs}^{\mathtt{i}}(\mathtt{j})$ to 1 if it needs data from processor j or to 0 if it does not. The scheduler then updates nmsgs on each processor with the element-by-element sum $\mathtt{nmsgs}^{\mathtt{i}}(\mathtt{j}) \leftarrow \sum_k nmsgs^k(j)$. This operation utilizes a function that imposes a fan-in tree to find the sums. At the end of the fan-in, on all processors, the entries of nmsgs are identical. The value $\mathtt{nmsgs}(\mathtt{j})$ is equal to the number of messages that processor $\mathtt{P}^{\mathtt{j}}$ must send during the exchange phase. In our example scenario, we see that at the end of the fan in, the value of nmsgs on each processor is [2,1,0] (Figure 6). Thus P_1 is able to determine that it needs to send data to two other (as yet unspecified) processors, P_2 needs to send data to one processor and P_3 does not need to send any data.

At this point, each processor transmits to the appropriate processor, a list of required array elements. This list contains the local offsets of the global array elements. In our example, P_1 sends a message to P_2 requesting

element 2 of the local array a'; P_2 and P_3 send a message to P_1 requesting element 2 of the local array a'. Each processor now has the information required to set up the send and receive messages that are needed to carry out the scheduled communications (Figure 7).

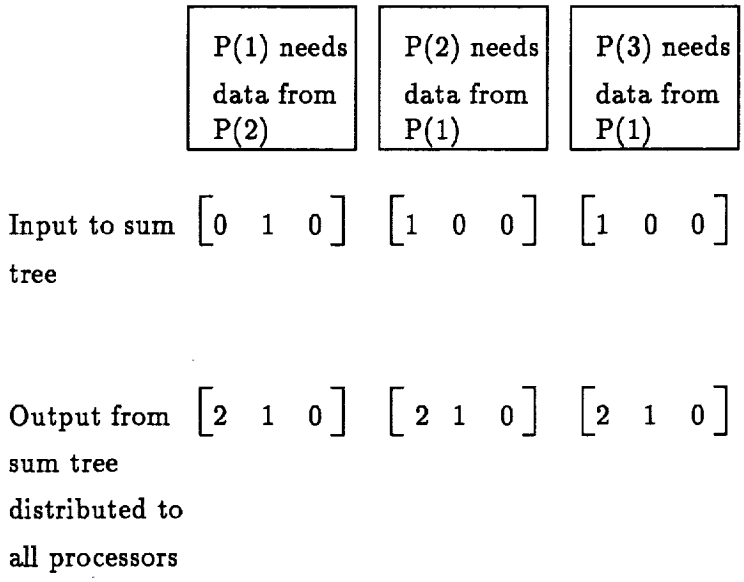

Figure 6: Computing the number of Send Messages

3.2 Data Exchange Primitives

Data *exchangers* can be called by each processor to:

gather data from other processors,

scatter data to other processors, or

perform global reduction operations

These exchangers use state information stored by the *scheduler*. As described in the previous section the *scheduler* determines the send and receive calls needed to carry out data exchanges. The scheduler is not given any

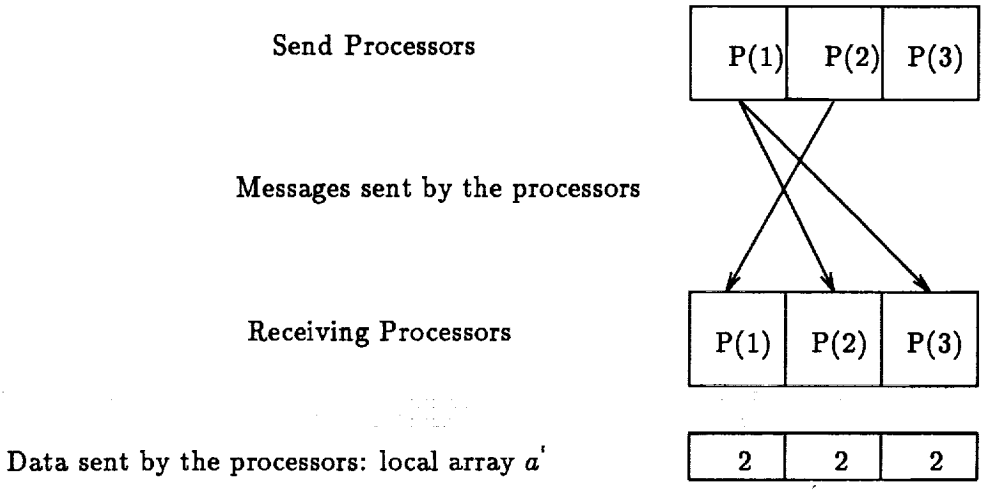

Figure 7: Final Message Pattern

information about memory locations – it involves only processors and local indices.

When a processor P calls a data exchanger, it passes to the *exchanger* routine the starting address of the first local array element in its memory. We call this address A_P. The *exchanger* routines use A_P to read or write distributed array elements. The schedule generated by the *scheduler* can be reused. A schedule can also be used to carry out identical patterns of data exchange on several different identically distributed arrays or on several different identically distributed array sections. The same schedule can be reused to repeatedly carry out a particular pattern of data exchange on a single distributed array, and any of the data exchange primitives can make use of a given schedule.

3.3 Calling Sequence of Scheduler and Data Exchanger

In this section, we give a specific example of the calling sequence used to invoke the schedule and data exchange primitives. We consider the following two Parti procedure calls:

call scheduler(id,n,hashed-cache,local-indices,processors)

call gather-exchanger(id,hashed-cache,local-array).

In this example, processor arranges to obtain copies of specified off-processor data elements, these copies are placed in the hash table hashed-cache.

Each processor passes to *scheduler* a list of off-processor local array indices. The scheduler will build a schedule that will make it possible for P to obtain n data elements. P will obtain data element i, $1 \leq i \leq n$ from processor $processors(i)$, local index $local - indices(i)$. A previously allocated hash table $hashed - cache$ is used to eliminate duplicate off-processor indices. *scheduler* returns an integer *id* which is be used by a subsequent call to *gather-exchanger*.

Each processor the calls *gather-exchanger*. *gather-exchanger* passes the address of the memory location $local - array$ in which each processor stores its portion of a distributed array. *gather-exchanger* returns copies of the requested off-processor array elements, these copies are placed in the hash table hashed-cache.

3.4 The Translation Table

We are able to allow a user to assign globally numbered distributed array elements to processors in an irregular pattern, using a distributed translation table. Recall that the *scheduler* and the data *exchangers* deal with indices of arrays that are *local* to each processor. The translation primitives, however, assume that distributed array elements have been assigned global indices.

The procedure *build-translation-table* constructs the distributed translation table. Each processor passes build-translation-table a set of globally numbered indices for which it will be responsible. The distributed translation table may be striped or blocked across the processors. With a striped translation table, the translation table entry for global index i is stored on processor $i \ mod \ numprocs$ where numprocs is the number of processors. In a blocked translation table, translation table entries are partitioned into a number of equal sized ranges of contiguous integers, these ranges are placed in consecutively numbered processors.

Dereference accesses the distributed translation table constructed in build-translation-table. For a given distributed array, dereference is passed a set of global indices that need to be located in distributed memory. Dereference returns the processors and memory locations where the specified global indices are stored.

Table 1: Translation Table Entries

global index	assigned processor	local index
Processor 1		
1	1	1
2	2	1
Processor 2		
3	2	2
4	1	2

Table 2: Results obtained from Dereference

global index	assigned processor	local index
Processor 1		
1	1	1
3	2	2
Processor 2		
2	2	1
3	2	2
4	1	2

We will illustrate the use of these primitives using the same mapping as in Figure 4 except that the number of processors equals 2. Two processors call build-translation-table. Thus P_1 claims responsibility for indices 1 and 4, while P_2 claims responsibility for indices 2 and 3. We assume that the translation table is partitioned between 2 processors by blocks. We depict the translation table data structure in Table 1. Each entry of the translation table assigns a processor and a local array index to each globally indexed distributed array element. In our example, translation table information about global indices 1 and 2 is stored in processor 1, while information about global indices 3 and 4 is stored in processor 2.

To continue our example, assume that both processors use the dereference primitive to find assigned processors and local indices corresponding to particular global distributed array indices. In Table 2 we depict the results obtained when processor 1 dereferences global indices 1 and 3, and processor 2 dereferences global indices 2, 3 and 4.

3.5 The Hashed Cache

The usefulness of the Parti primitives described in Section 3 can be enhanced by coupling these primitives with hash tables. The hash table records the numerical value associated with each distributed array element. The hash table also records the processor and local index associated with the element.

Dereference uses the hash table to reduce the volume of interprocessor communication. Recall that dereference returns the processor assignments and the memory locations that correspond to a given list of distributed array indices. Each distributed array index may appear several times in lists passed to dereference. The hash table is used to remove these duplicates.

The scheduler and the data exchange procedures use hash tables to store copies of off-processor distributed array elements. Lists of off-processor distributed array elements passed to the scheduler may have duplicates, the scheduler uses the hash table to remove these. The gather data *exchanger* (or *gather-exchanger*) fetches copies of off-processor distributed array elements and then *places the off-processor distributed array values in a hash table.* Similarly, *scatter-exchanger* obtains copies of off-processor distributed array elements *from a hash table* and writes the values obtained into a specified local array element on a designated processor. Primitives to support accumulations to non-local memory use hash tables in the same way scatter does.

Parti supplies a number of other primitives that support reading from, as well as writing and accumulating to, hash tables. When off-processor accumulations must be performed, we first carry out all possible accumulations to copies of distributed array elements in hash table, then we perform an accumulation data exchange.

Currently, we use a hash function that, for a hash cache of size 2^k, masks the lower k bits of the key. The key is formed by concatenating the processor-local index pair that corresponds to a distributed array reference.

4 The ARF Language

We have described in earlier sections the 2 distinct layers of the compiler. We will now briefly describe the extensions that we have added to Fortran 77 to create the ARF (ARguably Fortran) language. ARF is an interface between the application programs and the Parti run-time support primitives. The ARF compiler generates inspector and executor loops with embedded primitives.

Distributed arrays are declared in ARF source. These distributed arrays can either be partitioned between processors in a regular manner (e.g. equal sized blocks of contiguous array elements assigned to each processor), or in an irregular manner. An ARF user declares a mapping into distributed memory for each distributed array. When an array is to be partitioned in an irregular fashion, mapping information is specified in a regularly distributed integer array. Element `i` of the integer array describes the processor to which element `i` of the distributed array is to be mapped. Examples are shown below,

```
S1 distributed regular using block real k(SIZE)
S2 distributed regular using block integer map(SIZE)
S3 distributed irregular using map real y(SIZE).
```

S1 declares that `k` is a real array, distributed in a regular block manner, S2 declares that `map` is an integer array, also distributed in a regular block manner. S3 declares a real array `y` whose distribution is to be determined by the distributed integer array `map`. In the examples we give in this paper, all integer arrays used to specify irregular mappings were produced by hand coded partitioning procedures and then passed to an ARF routine.

Another addition to Fortran 77 is the *on clause*. The on clause has been originally implemented in Kali [14]. It is a mechanism by which the user has control over distributing the iteration space or work load among processors. *Distribute do* is an ARF language extension, this implies that the loop iterations in a given do loop should be distributed between processors. In the next section we use two examples to illustrate the transformations and optimizations performed by the ARF compiler. These message passing Fortran codes were generated by the ARF compiler.

4.1 Code Generation by the ARF Compiler

The ARF compiler transforms an ARF program into a target program which incorporates the primitives needed to efficiently carry out the distributed computation. The kernels we present here have been coded in ARF, compiled and run on an iPSC/860; in Section 5 we will present performance data obtained from both kernels.

4.1.1 Sparse Block Matrix Vector Multiply

In Figure 8 we present an ARF program that carries out a block sparse matrix vector multiply. This kernel was obtained from an iterative solver produced for a program designed to calculate fluid flow for geometries defined by an unstructured mesh [26]. The matrix is assumed to have size 4 by 4 blocks of non-zero entries. Statements S4 and S5 are loops that sweep over the non-zero entries in each block.

Integer array `partition` is local to each processor and enumerates a list of indices assigned to the processor. As mentioned earlier, the current implementation partitions only one dimension, the last dimension of the array. The Parti primitives, however, do support a broader class of array mappings [7]. Thus `partition` describes the partitioning of the last dimension of the arrays declared in statements S1 and S2. The ARF compiler uses the information in `partition` to make calls to primitives that initialize the distributed translation tables. These distributed translation tables are used to describe the mapping of `x`, `y` , `cols`, `ncols` and `f` (statements S1 and S2).

The partitioning of computational work is specified in statement S3 by an *on clause*. In this example, distributed array `partition` is used to specify which loop iterations are to be carried out on each processor. The reference `x(m,cols(j,i))` in S6 may require off-processor references. ARF must consequently generate an inspector to produce a schedule and a hash table to handle accesses to the distributed array `x`. A reference to the irregularly distributed array `f` occurs in statement S6. Note that distributed array `f` is irregularly distributed using array `partition` and that `partition` is also used by the on clause to partition loop iterations in S3. It can therefore be deduced that the reference to `f` in statement S6 is on-processor. `partition` specifies how distributed array elements and loop iterations are to be distributed between processors. A separate partitioning routine generates `partition`. In this paper, we simply assume that array `partition` is passed to the sparse matrix vector multiply kernel after having been generated elsewhere.

The ARF compiler generates an inspector and an executor to run on each processor. The work of the inspector is carried out on each processor as follows:

> Call *build-translation-table* using the mapping defined by array `partition`. Generate distributed translation table $T_{\texttt{partition}}$.

Call *dereference* to employ translation table $T_{\texttt{partition}}$ to find processor assignments, PA and local indices, LA for consecutive references to $x(m, cols(j,i))$.

Pass PA and LA to scheduler,generate schedule S.

Use PA and LA to setup hash table H.

The executor generated by ARF on processor P is depicted in Figure 9. In Figure 9 we use Fortran 90 notation where appropriate to enhance readability. Off-prococessor elements of x are gathered and placed in a hash table H (step I Figure 9). Values from x are obtained from H or from local memory as is appropriate (step IIa, Figure 9). Arrays PA and LA are used to distinguish local from off-processor array accesses. In step IIb, we accumulate to y. Note that the declarations in S1 and S3 in Figure 8 allow the compiler to determine that accumulations to y are local.

4.1.2 The Fluxroe Kernel

In section 5 we will present the ARF compiler output of a more complex kernel. This kernel is taken from a program that computes convective fluxes using a method based on Roe's approximate Riemann solver [27], [28]; we will call this kernel *fluxroe*. Fluxroe computes the flux across each edge of an unstructured mesh. Fluxroe accesses elements of array yold, carries out flux calculations and accumulates results to array y. As was the case in the sparse block matrix vector multiply kernel, four sections of each array are distributed and accessed in an identical manner. In Figure 10 we depict an outline of the fluxroe kernel. We denote the indices of the two vertices that comprise edge i by $n1 = n(i,1)$ and $n2 = n(i,2)$. To compute the fluxes $f(i,k)$ across the ith edge, we need to access $yold(n1,k)$ and $yold(n2,k)$, for $1 \leq k \leq 4$ (part I Figure 10). Once the fluxes have been computed, we add the newly computed flux values $f(i,k)$ to $y(n1,k)$ and subtract $f(i,k)$ from $y(n2,k)$ (part III Figure 10). Note that arrays y and yold are irregularly distributed using y-partition, and that distributed array node is irregularly distributed using edge-partition. Since the *on clause* in the distributed do statement also uses edge-partition to specify how loop iterations are to be partitioned, no off-processor references are made to node in part I Figure 10.

In the inspector, we need to compute a schedule S_{n1} for the off-processor *additions* to $y(n1,k)$ (part IIIa Figure 10), and a different schedule S_{n2} for

```
S1 distributed irregular using partition real*8 x(4,n), y(4,n),f(4,4,maxcols,n)
S2 distributed irregular using partition integer cols(9,n), ncols(n)

   ... initialization of local variables ...

S3 distributed do i=1,n on partition
      do j=1,ncols(i)
       S4 do k=1,4
             sum = 0
          S5 do m = 1,4
          S6 sum = sum + f(m,k,j,i)*x(m,cols(j,i))
             enddo
          y(k,i) = y(k,i) + sum
          enddo
      enddo
   enddo
```

Figure 8: ARF Sparse Block Matrix Vector Multiply

```
I.  call gather-exchanger using schedule S to obtain off-processor elements
    of x

    gather-exchanger places gathered data in hash table H

    count = 1

II. for all rows i assigned to processor P

    do j=1,ncols(i)

        do k= 1,4
            sum = 0
        IIa. if PA(count) == P then
                vx(1:4) = x(1:4,LA(count))
            else
                Use PA(count), LA(count) to get vx(1:4) from hash table
                H
            endif
            do m=1,4
                sum = sum + f(m,k,j,i)*vx(m)
            end do
        IIb. y(k,i) = y(k,i) + sum
        end do
        count = count + 1

    end do
```

Figure 9: Executor generated from ARF for Sparse Block Matrix Vector Multiply

the off-processor *subtractions* from $y(n2,k)$ (part IIIb Figure 10). When parallelized, fluxroe reads as well as accumulates to off-processor distributed array locations. As we pointed out in Section 3.2, any of the data exchange primitives can use the same schedule. We can use schedule S_{n1} to gather off-processor references from $yold(k,n1)$ (part Ia Figure 10), and we can use schedule S_{n2} to gather off-processor references from $yold(k,n2)$ (part Ib Figure 10).

The work of the inspector is carried out as follows:

Call *build-translation-table* using mapping defined by array y-partition. Generate distributed translation table $T_{y-partition}$.

Call *dereference* to employ translation table $T_{y-partition}$ to find:

1. Processor assignments $\mathtt{PA_{n1}}$ and local indices $\mathtt{LA_{n1}}$ for consecutive add accumulations to $y(k,n1)$ (the same $\mathtt{PA_{n1}}$ and $\mathtt{LA_{n1}}$ can be used for consecutive references to $y(k,n1)$).
2. Processor assignments $\mathtt{PA_{n2}}$ and local indices $\mathtt{LA_{n2}}$ for consecutive substract accumulations to $y(k,n2)$ (the same $\mathtt{PA_{n2}}$ and $\mathtt{LA_{n2}}$ can be used for consecutive references to $y(k,n2)$).

Pass $\mathtt{PA_{n1}}$ and $\mathtt{LA_{n1}}$ to *scheduler* to obtain schedule $\mathtt{S_{n1}}$; pass $\mathtt{PA_{n2}}$ and $\mathtt{LA_{n2}}$ to *scheduler* to obtain schedule $\mathtt{S_{n2}}$.

Setup hash tables $\mathtt{H_{n1}}$ and $\mathtt{H_{n2}}$.

Figure 11 outlines the executor produced by ARF on processor P. In Figure 11 we use Fortran 90 notation where appropriate to enhance readability. In step Ia and Ib we gather two sets of off-processor elements of yold using schedules S_{n1} and S_{n2}. In step II we access the appropriate elements of yold either from local memory or from the appropriate hash table, and in step III we use yold values to calculate fluxes. If the newly computed fluxes are to be accumulated to a local element of distributed array y, the appropriate addition or subtraction is carried out at once (steps IVa and IVc Figure 11). When a flux must be accumulated to a off-processor element of y, we accumulate the flux to *a copy of* y *stored in a hash table* (steps IVb and IVd Figure 11). When all fluxes have been calculated and all local accumulations are completed, we then call the *scatter-add* and *scatter-subtract* exchangers. These exchangers carry out the needed off-processor accumulations.

```
distributed irregular using y-partition real*8 yold(4,Number-nodes),
y(4,Number-nodes)

distributed irregular using edge-partition integer node(2,Number-edges)


... initialization of local variables ...


distributed do i = 1,Number-edges on edge-partition

  I. n1 = node(1,i)
     n2 = node(2,i)

     do k=1,4
     Ia. Va(k) = yold(k,n1)
     Ib. Vb(k) = yold(k,n2)
     end do

  II. Calculate flux using Va(k), Vb(k)

 III. do k=1,4
     IIIa. y(k,n1) = y(k,n1) + flux(k)
     IIIb. y(k,n2) = y(k,n2) - flux(k)
      end do
end do
```

Figure 10: ARF Kernel From Riemann Solver

The current version of the ARF compiler attempts to minimize the number of schedules to be computed. We might have produced a *single* schedule for all off-processor yold data accesses. If the inspector produced a single schedule for all accesses to yold, it would have been necessary to compute *three* different schedules in the inspector. Computing a single schedule for all references to yold might have led to a more efficient executor at the cost of a more expensive inspector.

4.2 Memory Utilization

We will give an overview of some of the memory requirements exacted by the methods described in this section, and suggest some ways in which these requirements can be reduced. Many sparse and unstructured programs use large integer arrays to determine reference patterns. In this respect, the kernels depicted here are typical. In Figure 8, a $9n$ element integer array cols is used for this purpose; while in Figure 10, a size $2 * Number - edges$ array node is employed. The executors depicted in Figure 9 and Figure 11 replace cols and node with local arrays that store the processor assignments and the local indices for references to irregularly distributed arrays. In the kernels in Figure 8, the sum of the number of elements used in all processors to store both processor assignments and local indices is no larger than $18n$; in Figure 10 the parallelized code uses a total of $4 * Number - edges$ elements.

The amount of additional storage needed for the parallelized code can be reduced in the following simple manner. The iterations I of a loop are divided into into two disjoint sets. The first set of iterations is $\mathtt{I}_{\mathtt{local}}$, where all memory references are to locally stored array elements. The second set is $\mathtt{I}_{\mathtt{off-processor}}$,in this set, each iteration contains some off-processor distributed array reference. In this case we need only to list processor assignments for loop iterations $\mathtt{I}_{\mathtt{off-processor}}$. Since it is frequently possible to map problems so that most memory references are local to a processor, a substantial memory savings will result.

The schemes described thus far would use very large quantities of extra memory when attempting to handle a loop in which a small number of distributed array elements are accessed many times. For instance, consider the following loop where f is a *function* defined so that $1 \leq f(i) \leq 2$ for any i.

Ia. call *gather-exchanger* using schedule S_{n1} to obtain first set of off-processor elements of yold
gather-exchanger places data in hash table H_{n1}.

Ib. call *gather-exchanger* using schedule S_{n2}, to obtain second set of off-processor elements of yold
gather-exchanger places data in hash table H_{n2}.

count = 1

II. for edges i assigned to processor P

if (PA_{n1}(count) .EQ. P) then
va(1:4) = yold(1:4,LA_{n1}(count)) else
get va(1:4) from hash table H_{n1}
endif
if (PA_{n2}(count) .EQ. P) then
vb(1:4) = yold(1:4,LA_{n2}(count)) else
get vb(1:4) from hash table H_{n2}
endif

III. Calculate fluxes flux(1:4) using va(1:4) and vb(1:4)

IV. if PA_{n1}(count) .EQ. P then
IVa. yold(1:4,LA_{n1}(count)) = yold(1:4,LA_{n1}(count)) + flux(1:4)
else
IVb. Accumulate flux(1:4) to hash table H_{n1}
endif
if PA_{n2}(count) .EQ. P then
IVc. yold(1:4,LA_{n2}(count)) = yold(1:4,LA_{n2}(count)) - flux(1:4)
else
IVd. Accumulate flux(1:4) to hash table H_{n2}
endif

end do
count = count+1

Va. Call *scatter-add exchanger* using schedule S_{n1} and hash table H_{n1}.

Vb. Call *scatter-subtract exchanger* using schedule S_{n2} and hash table H_{n2}.

Figure 11: Executor generated from ARF for Fluxroe Kernel

distributed irregular partition y

.....

do i=1, $HUGE - NUMBER$

.... y(f(i))

end do

In the above loop, the reference pattern of distributed array y is determined by f. In this example, at most *two* distinct elements of y are referenced in the loop. Loops of this sort can be handled by using a hash table to store processor and local index assignments for each *distinct* memory reference. In our example, each processor would have to store processor and local index assignments for no more than *two* references to distributed array y. There is a performance penalty that must be paid for using a hash table to find processor and local index assignments for distributed array elements. After examining a variety of sparse and unstructured codes, we chose not to implement the method described in this section in the ARF compiler. In [19], we present an analysis of the type of time and space tradeoffs outlined in this section.

5 Experimental Results

In this section we present a range of performance data that summarizes the effects of preprocessing on measures of overall efficiency and that gives some insight into the performance effects of problem irregularity and partitioning. Our computational experiments employed the fluxroe kernel and the block sparse matrix vector multiply kernel. Both kernels were coded in ARF; the parallelized benchmark numbers we present were obtained from programs generated by the ARF compiler. It should be noted that the syntax accepted by our ARF compiler differs in some minor ways from the that presented in the previous sections.

The experiments described in this paper used either a 32 processor iPSC/860 machine located at ICASE, NASA Langley Research Center or a 128 processor iPSC/860 machine located at Oak Ridge National Laboratories. Each processor had 8 megabytes of memory. We used the Greenhill 1.8.5 Beta version C compiler to generate code for the 80860 processors.

5.1 Unstructured Mesh Data

We use as input data a variety of unstructured meshes; both actual unstructured meshes obtained from aerodynamic simulations and synthetically generated meshes.

Unstructured Meshes from Aerodynamics : We use two different unstructured meshes generated from aerodynamic simulations.

Mesh A: A 21,672 element mesh generated to carry out an aerodynamic simulation involving a multielement airfoil in a landing configuration [17]. This mesh has 11143 points.

Mesh B: A 37,741 element mesh generated to simulate a 4.2 % circular arc airfoil in a channel [11]. This mesh has 19155 points.

Each mesh point is associated with an (x, y) coordinate in a physical domain. We use domain information to partition the mesh in three different ways; strips, orthogonal binary dissection algorithm ([5], [10]) and another mesh partitioning algorithm *jagged partitioning*, described in [24].

Synthetic Mesh from Templates

A finite difference template is used to link K points in a square two dimensional mesh. This connectivity pattern is incrementally distorted. Random edges are introduced subject to the constraint that in the new mesh, each point still requires information from K other mesh points.

This mesh generator makes the following assumptions:

I. The problem domain consists of a 2-dimensional square mesh of N points,

II. Each point is initially connected to K neighbors determined by a finite difference template,

III. With probability q, each mesh link is replaced by a link to a randomly chosen mesh point.

Note that when q is equal to 0.0, no mesh links are modified and no changes are introduced by step III. When q is equal to 1.0 we have a completely random graph. In this paper we will make use of two templates. One template connects each point to its four nearest neighbors (K=4); the other template connects each point to both its

four nearest neighbors as well as to each of its four diagonal neighbors (K=8). We refer to the $K = 4$ template as a *five point template* and we refer to the K=8 template as a *nine point template.* In the experiments to be described in the rest of this section, we employed a 256 by 256 point mesh.

5.2 Overall Performance

We first present data to give an overview of the performance we obtained on the iPSC/860 from the ARF compiler output. In the results depicted in this section, we use a blocked distributed translation table. In Table 3 we present a) the inspector time: time required to carry out the inspector preprocessing phase, b) computation time: the time required to perform computations in the iterative portion of the program and c) the communication time: the time required to exchange messages within the iterative portion of the program. The inspector time includes the time required to set up the needed distributed translation table as well as the time required to access the distributed translation table when carrying out the preprocessing in the inspector. Unstructured Meshes A and B were partitioned using orthogonal binary dissection. In these experiments, the ratio of the time required to carry out the inspector to the computation time required for a single iteration ranged from a factor of 0.7 to a factor of 3.6. Most of the preprocessing time goes to setting up and using the distributed translation table. For instance, consider the block matrix vector multiply on 64 processors using the 21,672 element mesh. The total preprocessing cost was 122 milliseconds, of which 111 milliseconds went to work related to the translation table. We define parallel efficiency for a given number of processors P as the sequential time divided by the product of the execution time on P processors times P. The sequential time was measured using a separate sequential version of the each kernel run on a single node of the iPSC/860. In Table 3 we depict under the column *single sweep efficiency*, the parallel efficiencies we would obtain were we required to preprocess the kernel each time we carried out the calculations. In reality, preprocessing time can be amortized over multiple mesh sweeps. If we neglect the time required to preprocess the problem in computing parallel efficiencies, we obtain the second set of parallel efficiency measurements; the *executor efficiency* presented in Table 3. The executor efficiencies for 64 processors ranged from 0.48 to 0.59, while the single sweep efficiencies ranged from 0.10 to 0.17.

In the experiments depicted in Table 3, the time spent computing is

at least a factor of 2 greater than the communication time. The executor efficiencies are, however, impacted by the fact that the computations in the parallelized codes are carried out less efficiently than those in the sequential program. The parallel code spends time accessing the hashed cache. It also needs to perform more indirections than does the sequential program.

Table 3: Performance on different number of processors

nprocs	inspector time(ms)	comp time(ms)	comm time(ms)	single sweep efficiency	executor efficiency
Sparse Block Matrix Vector Multiply - Mesh A					
32	148	49	9	0.15	0.55
64	122	25	9	0.10	0.48
Sparse Block Matrix Vector Multiply - Mesh B					
32	200	85	10	0.19	0.59
64	150	42	9	0.14	0.54
Fluxroe - Mesh A					
8	231	310	24	0.40	0.69
16	162	157	21	0.34	0.65
32	135	80	22	0.19	0.57
64	172	41	19	0.12	0.48
Fluxroe - Mesh B					
8	393	534	23	0.41	0.70
16	249	269	18	0.36	0.68
32	191	156	23	0.28	0.62
64	203	69	14	0.17	0.59

In Table 4, we investigate the performance of the fluxroe kernel for meshes with varying degrees of regularity and for varying mesh mappings. We used 32 processors in this experiment. In Table 4 we depict synthetic meshes derived from 5 and 9 point stencils with probability of edge move q equal to either 0.0 or 0.4. These meshes were mapped by 1-D strips or by 2-D blocks. As one might expect, for the synthetic meshes the communications costs increase dramatically for increasing q. We see these dramatic increases because both the volume of communication required and the number of messages sent per node are much higher for large q. Preprocessing costs also increased with q but while the communications costs went up by at least a factor of 16, preprocessing costs went up by at most a factor of 1.8.

We also depict in Table 4 results from Meshes A and B. We partitioned

Table 4: Performance on 32 processors with different meshes

nprocs	inspector time(ms)	comp time(ms)	comm time(ms)	single sweep efficiency	executor efficiency
5 point template synthetic mesh partioned into strips					
q=0.0	200	275	22	0.49	0.82
q=0.4	310	293	361	0.25	0.37
5 point template synthetic mesh partioned into 2-D block					
q=0.0	398	275	15	0.35	0.84
q=0.4	463	291	319	0.23	0.40
9 point template synthetic mesh partioned into strips					
q=0.0	211	583	21	0.58	0.80
q=0.4	385	620	530	0.31	0.42
9 point template synthetic mesh partioned into 2-D block					
q=0.0	447	589	20	0.46	0.79
q=0.4	595	624	527	0.28	0.42
Mesh A					
binary	134	80	22	0.24	0.57
jagged	135	81	22	0.24	0.56
strips	148	83	26	0.22	0.53
Mesh B					
binary	191	136	23	0.28	0.61
jagged	186	137	21	0.28	0.62
strips	219	149	31	0.24	0.54

the mesh in three different ways; strips, the orthogonal binary dissection algorithm and jagged partitioning. Both binary dissection and the jagged partitioning algorithm break the domain into two dimensional rectangular regions, and the two methods produce very similar performance results.

5.3 Breakdown of Inspector Overhead

In Table 5, we measure the cost of dereferencing and scheduling the fluxroe kernel on different numbers of processors. We again use a blocked translation table. We use a five point template and we partition the mesh either into 1-D strips or into 2-D blocks. When the mesh is partitioned into strips, dereference involves mostly local data accesses since the domain data and the translation table are identically partitioned. When strip partitioning is used, translation table initialization does not involve any communication. The measurements presented in Table 5 are defined in the following manner:

Executor time is the computation and communication time required to execute the kernel; it *does not include time required for preprocessing*,

Table initialization time is the time needed to initialize the distributed translation table,

Dereference time is the time taken by the *dereference* Parti primitive, and

Scheduler time is the time required to produce the communications schedule once the required processor locations and local indices have been found by *dereference*.

In Table 5 we note that the majority of the costs incurred by the inspector are due to the translation table initialization and dereference. For instance consider the case where 64 processors are used to carry out a sweep over a 2-D block partitioned mesh with a 5 point template. The translation table initialization and dereference together require 183 % of the executor time while the generation of the schedule requires only 12 % of the executor time.

In the problems depicted in Table 5, communication costs comprise a rather small fraction of the executor time, consequently the method used to partition the domain does not make a significant performance impact on executor time. In Table 5, the costs of translation table initialization and of dereference are both strongly dependent on how the domain is partitioned.

2-D block partitioning leads to higher translation table related costs, this is almost certainly due to the increased communication requirements needed for translation table initialization and dereference. Strip partitioning *per se* does not necessarily lead to low translation table related costs. In Table 4 we note that strip partitioning actually leads to higher inspector costs for both Mesh A and Mesh B. The translation table is partitioned so that blocks of contiguously numbered indices are assigned to each processor. However in Mesh A and Mesh B, mesh points are not numbered in a regular fashion so the indices corresponding to a domain strip are not contiguously numbered.

Table 5: Cost of dereferencing and scheduling on different number of processors

nprocs	executor time (ms)	table init time (ms)	dereference time (ms)	schedule time (ms)
5 point template synthetic mesh partioned into strips				
8	1192	131	143	83
16	606	115	109	42
32	297	92	83	27
64	167	63	62	17
5 point template synthetic mesh partioned into 2-D blocks				
8	1189	333	595	83
16	599	192	311	42
32	290	136	235	26
64	158	77	212	19

5.4 Cost of translation table

In Section 3.4 we described two straightforward ways to map a distributed translation table onto processors. We consider the question of how to distribute the translation table so as to minimize costs associated with translation table access. Table 6 compares the time required to carry out *dereference* on blocked and striped translation tables by depicting:

the time required to carry out a particular call to *dereference*,

the average number of non-local accesses to table entries required by *dereference*, and

the average number of non-local processors accessed during the call to *dereference*.

When we examine the results for unstructured Meshes A and B, we note no consistent performance difference in the cost required to dereference a blocked or a striped translation table. Similar numbers of off-processor table entries need to be accessed for either translation table distribution. Blocked translation tables do lead to superior performance when we use the synthetic meshes. For the reasons described in Section 5.3, we obtain particularly good results when we use a striped partition with a blocked translation table. It is of interest to note that the blocked translation table also proved to be superior when we used synthetic meshes partitioned in 2-D blocks.

Table 6: Cost of dereference on 32 processors

Problem	Indirect - Blocked			Indirect - Striped		
	Time (ms)	Nonlocal Data	Nonlocal Proc	Time (ms)	Nonlocal Data	Nonlocal Proc
Synthetic: 5 point template, strip partition						
q=0	109	256	1	346	2232	31
q=0.2	157	1045	17	365	2862	31
q=0.4	218	1825	17	368	3350	31
Synthetic: 5 point template, 2-D block partition						
q=0	235	2143	9	336	2078	31
q=0.2	326	2841	25	355	2782	31
q=0.4	330	3352	25	370	3273	31
Mesh A						
binary	97	768	21	96	743	31
jagged	98	772	20	98	751	31
strips	109	860	29	102	843	31
Mesh B						
binary	130	1271	24	122	1230	31
jagged	139	1293	24	130	1263	31
strips	159	1519	31	172	1513	31

5.5 Scheduler and Data Exchanger Performance

To quantify the communications costs incurred by the Parti scheduler and data exchange primitives, we measured the time required to carry out the *scheduler*, *gather-exchanger* and *scatter-exchanger* procedure calls and compared them to the hand coded version of iPSC/860 supplied sends and re-

Table 7: Overheads for Parti Scheduler and Gather-Exchanger Primitives

Number of Data Elements	Send Receive Time(ms)	Gather-Exchanger (ratio)	Scheduler (ratio)
100	0.5	1.0	2.1
400	1.0	1.1	1.4
900	1.8	1.1	1.3
1600	2.9	1.2	1.3
2500	4.3	1.2	1.1
3600	6.0	1.2	1.0

ceives; the sends and receives communicated the same amount of data as did the Parti procedures. We performed an experiment in which two processors repeatedly exchanged W single precision words of information. The exchange was carried out using gather-exchangers, scatter-exchangers and the iPSC/860 supplied send and receive calls. In Table 7 we depict the results of these experiments. We present the time (in milliseconds) required to carry out the requisite data exchange using send and receive messages. We then present the ratio between the time taken by the *scheduler* and *gather-exchanger* Parti primitive calls and the time taken by the equivalent send and receive calls. The *scatter exchanger* calls were also timed, the results of which were virtually identical to that of the corresponding *gather-exchanger* call.

¿From Table 7 we see that *gather-exchanger* took no more than 20% more time than explicitly coded send/receive pairs to move W words of information between two processors. The additional overhead required for *scheduler* to carry out the data exchange was a factor of 2.1 to 1.0 times the cost of using explicitly coded send/receive pairs to move W words.

6 Relation to Other Work

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], [16], [4], [29] and [10].

Several researchers have developed programming environments that are targeted towards particular classes of irregular or adaptive problems. Williams

[29] 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.

There are a variety of compiler projects targeted at distributed memory multiprocessors [30], [9], [21], [20], [1], [25]. With the exception of Kali project [15], and the Parti work described here and in [22], [18], and [23]; these compilers do not attempt to efficiently deal with loops that arise in sparse or unstructured scientific computations.

We have produced and benchmarked a prototype compiler that is able to generate code capable of efficiently handling kernels from sparse and unstructured computations. The procedures that carry out runtime optimizations are coupled to a distributed memory compiler via a set of compiler transformations. The compiler described and tested in this paper is qualitatively different from the efforts cited above in a number of important respects. We have developed and demonstrated mechanisms that allow us to support irregularly distributed arrays. Irregularly distributed arrays must be supported in order to make it possible to map data and computational work in an arbitrary manner. Because we can support irregularly distributed arrays, it was possible for us to compare the performance effects of different problem mappings (Section 5). Support for arbitrary distributions was proposed in [18] and [23] but to our knowledge, this is the first implementation of a compiler based distributed translation table mechanism for irregular scientific problems.

We find that many unstructured NASA codes must carry out data accumulations to off-processor memory locations. We chose one of our kernels to demonstrate this, and designed our primitives and compiler to be able to handle this situation. To our knowledge, our compiler effort is unique in its ability to efficiently carry out irregular patterns of off-processor data accumulations.

We augment our primitives with a hash table designed to eliminate duplicate data accesses. In addition, we use the hash table to manage copies of off-processor array elements. Other researchers have used different data structures for management of off-processor data copies [15].

7 Conclusion

In this paper we describe and experimentally characterize a compiler and runtime support procedures which embody methods that are capable of handling a wide range of irregular problems in scientific computing. After examining a number of complete NASA codes, we chose to demonstrate our methods using two kernels extracted from those codes. Both of these kernels involved computations over unstructured meshes. We coded both kernels in ARF, our dialect of Fortran, and generated code to ran on the nodes of the iPSC/860. Detailed timings were carried out on both kernels using unstructured meshes from aerodynamics, along with meshes that were generated by using random numbers to incrementally distort matrices obtained from a fixed finite difference template. This benchmarking suite stressed the communications capabilities of the iPSC/860 and the Parti primitives in a variety of ways.

In the experiments we reported in Section 5.2, we saw that that the ratio of the time required to carry out all preprocessing to the time required for a single iteration of either kernel ranged from a factor of 0.7 to a factor of 3.6. We then saw in Section 5.3 that the majority of the preprocessing costs arose from the need to support irregularly distributed arrays. In Section 5.5 the performance of the *scheduler* and *data exchanger* Parti primitives were quantified. The *data-exchangers* turned out to be at most 20% more time consuming than the analogous send and receive calls provided by Intel.

We believe that one of the virtues of our layered approach to distributed compiler design is that we have managed to capture a set of critical optimizations in our runtime support primitives. Our primitives, and hence our optimizations, can be migrated to a variety of compilers targeted towards distributed memory multiprocessors. We intend to implement these primitives in the ParaScope parallel programming environment [13]. In addition, Parti primitives can and are being used directly by programmers in applications codes [7].

Most of the complexity of our system is in the Parti procedures. The Parti procedures have been developed so that that transformations needed to embed the appropriate primitives can be implemented with relative ease in distributed memory compilers. The primitives used to implement the runtime support include communications procedures designed to support irregular patterns of distributed array access, and procedures to find the

location of irregularly mapped distributed array data using distributed translation tables. Primitives also support the maintenance of hash tables used to store copies of off-processor data.

8 Acknowledgements

We would like to thank Harry Jordan and Bob Voigt for their careful editing of this manuscript. We would also like to thank the Advanced Computing Laboratory at Oak Ridge National Laboratories and NAS at NASA Ames for providing us access to their 128 node Intel iPSC/860 hypercubes.

We wish to thank Dimitri Mavriplis and David Whitaker for supplying us with unstructured meshes, and to David Whitaker and P Venkatkrishnan for access to their codes.

References

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

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

[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, January 1988, pp. 1698,1711.

[5] M. J. Berger and S. H. Bokhari, *A partitioning strategy for pdes across multiprocessors*, in The Proceedings of the 1985 International Conference on Parallel Processing, August 1985.

[6] H. Berryman and J. Saltz, *A manual for parti runtime primitives*, Interim Report 90-11, ICASE, 1990.

[7] H. Berryman, J. Saltz, and J. Scroggs, *Execution time support for adaptive scientific algorithms on distributed memory machines*, Report 90-41, ICASE, May 1990.

[8] S. Bokhari, *Communication overhead on the intel ipsc-860 hypercube*, Report 90-10, ICASE Interim Report, 1990.

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

[10] 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.

[11] *Numerical methods for the computation of inviscid transonic flows with shock waves - a gamm workshop*, in Notes on Numercial Fluid Mechanics, vol. 3.

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

[13] 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.

[14] C. KOELBEL AND P. MEHROTRA, *Compiling global name-space programs for distributed execution*, Report 90-70, ICASE, 1990.

[15] C. KOELBEL, P. MEHROTRA, AND J. V. ROSENDALE, *Supporting shared data structures on distributed memory architectures*, in 2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming, ACM SIGPLAN, Mar. 1990, pp. 177–186.

[16] J. W. LIU, *Computational models and task scheduling for parallel sparse cholesky factorization*, Parallel Computing, 3 (1986), pp. 327–342.

[17] D. J. MAVRIPLIS, *Multigrid solution of the two-dimensional Euler equations on unstructured triangular meshes*, AIAA Journal, 26 (1988), pp. 824–831.

[18] R. MIRCHANDANEY, J. H. SALTZ, R. M. SMITH, D. M. NICOL, AND K. CROWLEY, *Principles of runtime support for parallel processors*, in Proceedings of the 1988 ACM International Conference on Supercomputing , St. Malo France, July 1988, pp. 140–152.

[19] S. MIRCHANDANEY, J. SALTZ, P. MEHROTRA, AND H. BERRYMAN, *A scheme for supporting automatic data migration on multicomputers*, in Proceedings of the Fifth Distributed Memory Computing Conference, Charleston S.C., 1990.

[20] A. ROGERS AND K. PINGALI, *Process decomposition through locality of reference*, in Conference on Programming Language Design and Implementation, ACM SIGPLAN, June 1989.

[21] M. Rosing, R. Schnabel, and R. Weaver, *Expressing complex parallel algorihtms in dino*, in Proceedings of the 4th Conference on Hypercubes, Conurrent Computers and Applications, 1989, pp. 553–560.

[22] J. Saltz and M. Chen, *Automated problem mapping: the crystal runtime system*, in The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN, September 1986.

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

[24] J. Saltz, S. Petiton, H. Berryman, and A. Rifkin, *Performance effects of irregular communications patterns on massively parallel multiprocessors*, Report 91-12, ICASE, 1991.

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

[26] P. Venkatkrishnan, J. Saltz, and D. Mavriplis, *Parallel preconditioned iterative methods for the compressible navier stokes equations*, in 12th International Conference on Numerical Methods in Fluid Dynamics, Oxford, England, July 1990.

[27] D. L. Whitaker and B. Grossman, *Two-dimensional euler computations on a triangular mesh using an upwind, finite-volume scheme*, in Proceedings AIAA 27th Aerospace Sciences Meeting, Reno, Nevada, January 1989.

[28] D. L. Whitaker, D. C. Slack, and R. W. Walters, *Solution algorithms for the two-dimensional euler equations on unstructured meshes*, in Proceedings AIAA 28th Aerospace Sciences Meeting, Reno, Nevada, January 1990.

[29] R. D. Williams and R. Glowinski, *Distributed irregular finite elements*, Tech. Rep. C3P 715, Caltech Concurrent Computation Program, February 1989.

[30] H. ZIMA, H. BAST, AND M. GERNDT, *Superb: A tool for semi-automatic MIMD/SIMD parallelization*, Parallel Computing, 6 (1988), pp. 1–18.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-187515 ICASE Report No. 91-13		

4. Title and Subtitle

DISTRIBUTED MEMORY COMPILER DESIGN FOR SPARSE PROBLEMS

5. Report Date

January 1991

6. Performing Organization Code

7. Author(s)

Janet Wu
Seema Hiranandani
Joel Saltz
Harry Berryman

8. Performing Organization Report No.

91-13

10. Work Unit No.

505-90-52-01

9. Performing Organization Name and Address

Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

11. Contract or Grant No.

NAS1-18605

13. Type of Report and Period Covered

Contractor Report

12. Sponsoring Agency Name and Address

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Michael F. Card

Submitted to IEEE Transactions Software Engineering

Final Report

16. Abstract

In this paper we describe and demonstrate a compiler and runtime support mechanism. The methods presented here are capable of solving a wide range of sparse and unstructured problems in scientific computing. The compiler takes as input a Fortran 77 program enhanced with specifications for distributing data, and the compiler outputs a message passing program that runs on a distributed memory computer. The runtime support for this compiler is a library of primitives designed to efficiently support irregular patterns of distributed array accesses and irregular distributed array partitions. We present a variety of Intel iPSC/860 performance results obtained through the use of this compiler.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
distributed memory, unstructured grids, sparse, compilers	60 - Computer Operations and Hardware 61 - Computer Programming and Software Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	40	A03