4-1992

Embedding Data Mappers with Distributed Memory Machine Compilers

Ravi Ponnusamy
Syracuse University

Joel Saltz

Raja Das

Charles Koelbel

Alok Choudhary
Syracuse University

SU-CIS-92-06

Embedding Data Mappers with Distributed Memory Machine Compilers

Ravi Ponnusamy, Joel Saltz, Raja Das,
Charles Koelbel, and Alok Choudhary

April 1992

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

Embedding Data Mappers with Distributed Memory Machine Compilers

Ravi Ponnusamy[1], Joel Saltz[2], Raja Das[2]
Charles Koelbel[3] and Alok Choudhary[1]

[1] Northeast Parallel Architecture Center, Syracuse University, Syracuse, NY 13244
[2] ICASE, MS 132C, NASA Langley Research Center, Hampton, VA 23666
[3] Center for Research on Parallel Computation, Rice University, Houston, TX 77251

1 Introduction

In scalable multiprocessor systems, high performance demands that computational load be balanced evenly among processors and that interprocessor communication be limited as much as possible. Compilation techniques for achieving these goals have been explored extensively in recent years [3, 9, 11, 13, 17, 18]. This research has produced a variety of useful techniques, but most of it has assumed that the programmer specifies the distribution of large data structures among processor memories. A few projects have attempted to automatically derive data distributions for regular problems [12, 10, 8, 1]. In this paper, we study the more challenging problem of automatically choosing data distributions for irregular problems.

By irregular problems, we mean programs in which the pattern of data access is input-dependent, and thus not analyzable by a compiler. For example, the loop

```
do i = 1, nedges
   n1 = nde(i,1)
   n2 = nde(i,2)
   y(n1) = y(n1) + x(n1) + x(n2)
   y(n2) = y(n2) - x(n1) + x(n2)
enddo
```

sweeps over the edges of an unstructured mesh. This is a simplified version of a common loop type in unstructured mesh computational fluid dynamics algorithms. The array nde is assigned at execution time, thus severely limiting the compiler analysis that is possible. Efficiently executing

this loop requires partitioning the data and computation to balance the work load and minimize communication. As the information necessary to evaluate communication (i.e. the contents of nde) are not available until runtime, this partitioning must be done on the fly. Thus, we focus on runtime mappings in this paper.

Over the past few years a lot of study has been carried out in the area of mapping irregular problems onto distributed memory multicomputers. As a result of this, several general heuristics have been proposed for efficient data mapping [2, 4, 6, 7, 16]. But currently these partitioners must be coupled to user programs manually. In this paper we describe an automatic method for linking these partitioners. We first describe how irregular data mapping heuristics can be *linked* to distributed memory compilers by generating a standard distributed data structure. We then describe the Parti mapper coupling primitives that can be used to generate the standard structure and to link partitioners. We then show how a compiler can produce code to generate this structure at run-time, thus effectively removing the responsibility of data partitioning from the programmer. The work described here is being pursued in the context of the CRPC Fortran D project [9].

2 Compiler Embedded Run-time Mapping: Strategy

2.1 Problem Statement

Our goal is to allow automatic linkage of partitioning heuristics which use as their main input the connectivity of the major data structures. As we describe in the next section, the solution to this problem requires new compiler directives, compiler transformations, and a run-time environment. This paper emphasizes the run-time environment issues; we will detail the other parts in other work.

In many scientific codes, most of the work consists of computing data values of many elements connected through a run-time data structure such as a tree or directed graph. A good example is sparse matrix computations, in which the sparsity structure of a matrix is often represented as a graph; the computational work is in computing matrix values at each node in that graph. We will

always consider the underlying structure to be a graph, since any other interconnected structure can be considered a special type of graph. The partitioners that we consider are based on finding a division of this underlying graph which "breaks" as few edges as possible. An edge is "broken" when the elements it connects are allocated to different processors; in this case, communication is needed to bring the values together for the computation.

To make our implementation tractable, we constrain the types of problems that we will partition. We limit ourselves to array partitioning based on loops in which all distributed data arrays conform in size and are to be identically distributed. We also restict ourselves to FORALL loops, that is, loops for which the iterations can safely be executed in any order. Different loop iterations may access the same memory location only if all accesses are reads, or if the accesses are an accumulation using a commutative and associative operator (such as a summation). We also assume that entire statements are executed on one processor, rather than computing subexpressions on different processors as is sometimes done on SIMD machines. The statements in a single loop iteration may, however, be executed on different processors under the "owner computes" model explained later.

2.2 General Strategy

Our approach to mapping irregular problems has three components:

1. The programmer inserts compiler directives to mark the important loops that will determine partitioning. Generally, these will be loops over the main data structure in the program, where we assume most of the computation occurs. These are the most important loops to optimize because of the time they require and because they are likely to generate most of the communication in a program.

2. The compiler generates run-time code to perform several phases of analysis based on the marked loops. The compiler cannot perform the required analysis directly, because it depends on data that is only available during actual execution. Instead, the marked loops are modified

to generate a data structure with the required information; the modified versions of the loops will be executed at run time. This technique was previously used by the Kali [11] and PARTI [15] projects to implement communications for irregular problems; here, we extend that work to generating data and computation partitionings.

3. At run time, the generated code is executed, producing data structures that can be input to the partitioners. Run-time environment support is needed for all of generating the data structures, feeding them to the partitioner, and implementing the resulting partition. This support handles both the computation of maintaining the data structures and the communication of sharing locally necessary parts of the structure between processors. We have implemented the run-time environment as a series of enhancements to PARTI, a system designed specifically for implementing irregular computations on distributed-memory computers.

The structure of all three components is closely tied to the class of partitioning strategies used. We have chosen a graph-based approach desribed below; other approaches based on problem geometry or domain-specific information are also possible. We could incorporate these approaches by adding annotations and compiler transformations which extract the input needed for these partitioners.

The partitioning scheme we use has two stages:

1. Given the array accesses made by a program, determine a good partitioning of the data.

2. Given a data partitioning and a loop, partition the iterations of a loop among processors.

Each of these stages uses a graph-based data structure.

To implement the first stage, we use a distributed data structure called the *Runtime Data Graph* or **RDG**.[1] In brief, this is a directed graph telling, for each array element, which other elements are used to compute it. The **RDG** thus represents the loop's computational pattern. It is the first-

[1] In previous papers we have referred to this structure as the Runtime Dependence Graph. Unfortunately, "dependence" has a specific meaning in the compiler literature that is incompatible with the **RDG**'s meaning. Since we need the compiler concept in other work, we have changed our notation slightly.

stage partitioner's task to divide this graph to maximize the number of linked elements mapped to the same processor while balancing the memory usage among processors.

In the second stage, there are two possibilities for using this mapping. We could assign work to processors using the "owner computes" rule; that is, the processor that owns the left-hand expression of an assignment is responsible for computing the right-hand side. This requires no new graph to be generated, but may involve substantial computation to determine which processor is to execute each statement. Alternately, we can assign computational work to processors by executing all computations in a given loop iteration on one processor. To do this while taking advantage of the data partition computed above, we generate a distributed data structure we call the *Runtime Iteration Graph* or **RIG**. The **RIG** describes which distributed array elements are accessed during each loop iteration. The task of the second partitioner is to maximize the number of local elements accessed by all iterations while balancing the computational load.

The next section provides a more detailed look at how these data structures are represented and used.

3 Compiler Embedded Run-time Mapping: Implementation

In this section we describe how the Runtime Data Graph and Runtime Iteration Graph are implemented and generated. Bear in mind that the code for these operations is generated by the compiler but executed at runtime. Therefore, the data structures and operations must be executed in parallel in order for the system to have acceptable performance.

3.1 The Runtime Data Graph

The fundamental question that arises in implementing an irregular algorithm on a distributed memory machine is how the data will be distributed. All later decisions are influenced by the data distribution, and the overall efficiency of the program is likely to be determined by the quality of the distribution. In our system, the distribution is determined automatically using graph-theoretical

techniques.

The Runtime Data Graph (**RDG**) records the linkages between between elements of an array. The intent is that two elements will be linked if one is used to compute the other. More formally, an **RDG** is an undirected graph with one node for each distributed array element. Since we assume that all distributed arrays are to be partitioned in the same way, node i of the **RDG** represents element i of all distributed arrays if there is more than one. There is an edge between nodes i and j if, on some iteration of the loop, an assignment statement writes element i of an array (i.e. $x(i)$ appears on the on the left-hand side) and references element j (i.e. $y(j)$ appears on the right-hand side), or vice-versa. Edges are weighted by the frequency with which elements are linked.

A **RDG** is constructed by executing a modified version of the marked loop which forms a list of edges instead of performing numerical calculations. The modified loop executes in parallel by dividing the loop iterations in a fixed pattern and forming a local list on each processor. This loop adds edges $\langle i, j \rangle$ and $\langle j, i \rangle$ to the local list when a reference to array index i appears on the left side of a statement and a reference to j appears on the right side. Each time edge $\langle i, j \rangle$ is encountered, the modified loop increments a counter associated with $\langle i, j \rangle$. Self-loops (i.e. edges of type $< i, i >$) are ignored in the graph generation process as they will not affect partitioning. Each processor then flattens its local list into an adjacency list data structure closely related to Saad's Compressed Sparse Row (CSR) format [14]. A global scatter operation (resolving collisions by appending lists) is then used to combine these local lists into a complete graph, also represented in CSR format. This data structure is distributed so that each processor stores the adjacency lists for a subset of the nodes. (A fixed initial distribution is used at this point to divide the data.) This is the data structure passed to the **RDG** partitioner. The output of the partitioner is a distributed translation table [5, 15] describing an irregular mapping The basic PARTI primitives are then used to associate this table with each of the identically distributed arrays referenced in the loop.

3.2 The Runtime Iteration Graph

Once we have partitioned data, we must partition computational work. If we use the "owner computes" convention, it is clear how work will be partitioned. Otherwise, as we describe below, we must divide each loop's iterations among processors using a new partitioner.

To partition loop iterations, we use the Runtime Iteration Graph, or **RIG**. The **RIG** associates with each loop iteration all indices of each distributed array accessed during that iteration. More formally, the **RIG** is a bipartite directed graph consisting of two types of nodes. There is one node for every distributed array element, as was the case in the **RDG**; these nodes can only be sources of edges. There is also one node for each iteration of the loop; these nodes are always sinks of graph edges. There is an edge from array element i to iteration j if iteration j writes to or reads from distributed array element $x(i)$.

The **RIG** is generated for every loop that references at least one irregularly distributed array or that accesses any array in an irregular manner (e.g. through an indirection). Its generation is somewhat simpler than the **RDG**'s. Each processor is assigned a subset of the iterations by some simple scheme. The processor then generates the Compressed Sparse Row format of its local subgraph by executing its iterations and listing all distributed array references on each iteration as they are encountered. From this we can derive the *Runtime Iteration Processor Assignment graph* (**RIPA**) which lists, for each loop iteration, the number of distinct data references to array elements stored on each processor. This information can be generated by running through the **RIG**, checking array element locations in the distributed translation table. Our current mapper inputs the **RIPA** and assigns iterations to processors by putting each iteration on the processor where it accesses the most data. In the future, we plan to move toward more sophisticated partitioners which also take load balancing into consideration.

3.3 Run-time Environment Support

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

The **RDG** generation phase starts with an an initial distribution of loop iterations. In many cases this distribution, I_{init}, will be a simple default distribution. In some situations (e.g. adaptive codes), preprocessing to support irregular array mappings may have already been carried out. Our runtime support will handle either regular or irregular initial loop iteration distributions I_{init}.

RDG generation uses the following mapper coupling PARTI procedures. Procedure *eliminate_dup_edges* uses a hash table to store unique directed **RDG** edges, along with a count of the number of times each edge has been encountered. This implements insertion into the edge list as described above. We define the *local* loop **RDG** as the restriction of the loop **RDG** to a single processor. The local loop **RDG** includes only distributed array elements associated with I_{init}. Once all edges in a loop have been recorded, *generate_RDG* generates the local loop **RDG** (in Compressed Sparse Row form) and then merges all local loop **RDG** graphs to form the global loop **RDG**.

The data structures that describe the loop **RDG** graph are passed to a data partitioner *RDG_partitioner*. *RDG_partitioner* returns a pointer to a distributed translation table that describes the new mapping. (Note that *RDG_partitioner* can use any heuristic to partition the data, the only constraint is that the partitioners have the correct calling sequence. We have one partitioner working now, and plan to experiment with others.) Once the partitioner generates an efficient mapping the data can be remapped by using the procedure *remap*. Procedure *remap* is passed a pointer to the distribution translation table of the old data distribution and a pointer to the distribution translation table of the current data distribution . It returns a pointer to a schedule [5] which can be used by the PARTI *gather* routine for remapping the data.

No special support is needed for building the **RIG**, since it is a purely local operation on relatively simple data structures. The partitioning of loop iterations is supported by two primitives,

deref_rig and *iter_partition.* The **RIG** is generated by code transformed by a compiler. The primitive *deref_rig* takes the **RIG** as input. This primitive accesses distributed translation tables to find the processor assignments associated with each distributed array reference. *deref_rig* returns the **RIPA**. The **RIPA** is partitioned using the iteration partitioning procedure, *iter_partition.*

3.4 Compiler Support

In Fortran D, a user declares a template called a ***distribution pattern*** that is used to characterize the significant attributes of a distributed array. The distribution fixes the size, dimension and way in which the array is to be partitioned between processors. A distribution is produced using two declarations. The first declaration is *DECOMPOSITION.* Decomposition fixes the name, dimensionality and size of the distributed array template. The second declaration is *DISTRIBUTE.* Distribute is an executable statement and specifies how a template is to be mapped onto processors. Fortran D provides the user with a choice of several regular distributions. In addition, a user can create irregular distributions by giving a processor "home" for every decomposition element, using either an integer array or an integer-valued function. A specific array is associated with a distribution pattern using the Fortran D statement *ALIGN.*

In this project, we have extended the Fortran D syntax to implicitly specify processor mapping in a *DISTRIBUTE* statement by refering to a labelled loop and to a choice of partitioner. The current Fortran D syntax allows the user to force all work in an individual loop iteration to be assigned to a single specified processor by using the *ON clause* [11, 9]. The default compiler strategy is to use the owner-computes rule for loops. Our new syntax makes it possible for a user to specify what method will be used to partition loop iterations.

We introduce a new compiler directive namely, MAPLOOP. The directive MAPLOOP allows the user to specify the loop to be used for data partitioning. Normally, the user will choose the most computationally intensive loop in the program. The directive also specifies one or more conformal arrays based on which RDG (as discussed in Section 3.1) should be generated. For example, in

```
      call precoditioner...
      convergence test...
      MAPLOOP (kp,p) matmul
      COMPUTE ON MAXOWNER (kp(j))
C       Sparse Matrix Vector Multiplication (SPMVM)
      do j = 1, nrows
S1          kp(j) = kp(j) + diags(j)*p(j)
          do k = ptr(j), ptr(j+1)
S2            kp(j) = kp(j) + p(cols(k)) * vals(k)
S3            kp(cols(k)) = kp(cols(k)) + p(j)*vals(k)
          end do
      end do
      call saxpy...
      call saxpy...
      call precoditioner...
      convergence test...
```

Figure 1: A Skeleton Conjugate Gradient Solver Code With Compiler Directives

Figure 1 the directive specifies that the loop *matmul* should be selected for partitioning. Further, it specfies that RDG should be generated using access patterns of arrays *kp* and *p*.

Once arrays are partitioned, the computation should be assigned evenly across processors such that the volume of communication is reduced. The compiler directive COMPUTE ON MAXOWNER can be used to partition loop iterations. A particular loop iteration is assigned to the processor that has the maximum number of array elements specified in the array-list of the directive. For example, in Figure 1 the directive informs the compiler that loop iteration j should be assigned to the processor to which $kp(j)$ is assigned. On the other hand, if Figure 2, each loop iteration i is assigned to a processor that contains the maximum number of x and y array elements for that loop iteration. This minimizes the volume of communcation for each loop iteration. If there was tie in processor assignment it would be resolved arbitrarily.

4 Parti Mapper Coupling Primitives

In this section we illustrate how to use the PARTI data mapping and remapping primitives manually. We explain the primitives with the help of a conjugate gradient solver code and an Euler solver

```
MAPLOOP (x, y) edges
COMPUTE ON MAXOWNER (x(n1), x(n2),y(n1),y(n2))
do i = 1, nedges
  n1 = nde(i,1)
  n2 = nde(i,2)
  y(n1) = y(n1) + x(n1) + x(n2)
  y(n2) = y(n2) - x(n1) + x(n2)
enddo
```

Figure 2: A Sweep over Edges Code With Compiler directives

code. Figure 1 shows a skeleton code of a diagonally preconditioned conjugate gradient solver. The sparse matrix vector multiplication(SPMVM) is most the computationally intensive part of this solver. Hence, to achieve overall good performance, the SPMVM must be efficiently implemented. The performance of the SPMVM depends on the mapping of arrays *kp* and *p* accessed in SPMVM.

To link partitioners with the solver, the RDG must be generated first. The access patterns of the distrubted arrays *kp* and *p* can be represented as a graph (RDG) using three Parti data mapping primitives - 1) init_rdg_hash_table 2) eliminate_dup_edges and 3) generate_rdg. As a first step to generate RDG, the local RDG is generated using the primitive eliminate_dup_edges. The local RDG is stored in a hash table. The hash table for local RDG is initialized using a primitive init_rdg_hash_table as shown in Figure 3. The local loop iteration size *n_local* of the selected loop is an approximate initial value for allocating space for the hash table.

Initially, to generate RDG the loop iterations are evenly divided among processors. For each statment containing both read and write access to the distributed arrays, the primitive eliminate_dup_edges is called to generate local RDG. The primitive eliminate_dup_edges constructs local RDG using the hash table. In Figure 1, statements S1, S2 and S3 access the distributed array both on left and right side. However, the primitive eliminate_dup_edges is called only for the statement S2 because of the following reasons. For each iteration j, S1 accesses same index of the distributed arrays for the same iteration j. The statement S3 access the same set of indices($cols(k)$ and j) as that of the statement S2 for each loop iteration j. For each local iteration i, the primitive

eliminate_dup_edges takes in

- a pointer to the hash table *hashindex*,
- the left hand side global index j of iteration i,
- the list of left hand side global indices $cols(j)$ of the distributed arrays for iteration i, and
- the number of right hand side indices accessed (*n_dep*).

Once the local graph is constructed, it is merged using the primitive generate_rdg to form a distributed graph. This procedure converts the local graph from hash table (generated by eliminate_dup_edges) format to the CSR format. This primitive collects all those indices that share an edge to its initial distribution (*local_rows*) of array indices. This procedure takes in

- pointer to the hashtable *hashindex*,
- initial local iterations indices (*local_rows*)(note that the loops iterations are divided in an even distribution and
- The number of local iterations (*n_local*) and

and returns

- distributed **RDG** in CSR format (*csr_ptr* and *csr_cols*).

The graph representation of the distributed arrays (*kp* and *p*) is passed to the procedure RDG_parititioner. This procedure would use one of the data mapping heuristics proposed in [16, 7]. The parallel mapper returns a pointer to a translation table (*ttable*) which describes the distribution of arrays *kp* and *p*.

Once the arrays have been distributed, it is also necessary to efficiently distribute the loop iterations to reduce communication. The iteration partitioner is called to distributes the loop iterations to balance the computation and to reduce off-processor memory accesses.

```
      n_dep = 1
      ncount = 1
      hashindex = init_rdg_hash_table(n_local)
      do j = local_rows(1), local_rows(n_local)
C        kp(j) = kp(j) + diags(j)*p(j)
        do k = ptr(j), ptr(j+1)
C          kp(j) = kp(j) + p(cols(k)) * vals(k)
          f = j
          g = cols(k)
          call eliminate_dup_edges(hashindex, f, g, n_dep)
C           kp(cols(k)) = kp(cols(k)) + p(j)*vals(k)
        end do
      end do
      call generate_rdg(hashindex, local_rows, n_local, csr_ptr, csr_cols)
      call RDG_partitioner(local_rows, nlocal, csr_ptr, csr_cols, ttable)
```

Figure 3: Pre-processing Code for SPMVM Kernel for Data Mapping

Consider the sweep over edges, shown in Figure 2, of a typical fluid dynamics code. Assuming that the arrays x and y in the code have been distributed already the primitived dref_rig and iteration_partitioner can be used to partition the loop iterations. The primtive dref_rig takes in

- The array indices of the distributed arrays (*rig*) for each iteration i,
- The number of different indices of distributed arrays referred in each iteration (*n_ref*),
- The total number of loop iterations (*icount*),

and returns

- The processor assignment graph (*ripa*).

At run-time, the list of processor numbers in *ripa* is calculated by dereferencing the array distribution information stored in *ttable*. The list of processor numbers *ripa* is passed to the primitive iteration_partitioner. This primitive returns

- a list of local iterations (*iter_list*) and
- the number of local iterations (*n_iter*).

```
      icount = 1
      n_ref = 1
      do i = local_edge(1), local_edge(nedges)
        n1 = nde(i,1)
        n2 = nde(i,2)
C         y(n1) = y(n1) + x(n1) + x(n2)
        rig(icount, 1) = n1
C         y(n2) = y(n2) - x(n1) + x(n2)
        rig(icount, 2) = n1
        icount = icount+1
      end do
      dref_rig(ttable, rig, n_ref, icount, ripa)
      iteration_partitioner(ripa, n_ref, icount, iter_list, n_iter)
```

Figure 4: Pre-processing Code for Sweep Over Edges for Iteration Partitioner

The primitive iteration_partitioner assigns an iteration i to a processor which has the maximum number of arrays indices referred in iteration i.

In some cases, it is enough to derive the loop iteration paritions based on array distribution. For instance, in the case of SPMVM shown in Figure 1 the loop iterations can be distributed based on the distributions of arrays *kp* or *p*.

The arrays distributed initially in a known regular fashion (block) among processors can be remapped according to the new mapping. The primitive csr_remap can be used for this purpose. This primitive returns two schedules; the first can be used to remap data in CSR format and the second schedule can be used to map arrays conforming to the current mapping. The remapping can be carried out using the PARTI gather routines [5]. For example, in the case of SPMVM the arrays *vals* and *diags* are initially mapped to processors in a known manner. After the distribution of arrays *kp* and *p* the array *diags* must be remapped using the first schedule and the array *daigs* must be mapped using the second schedule returned by the primitive csr_remap. The schedule The remapping primitive csr_remap takes in

- The translation table representing the new mapping,
- the initial array distribution in CSR format - a list of indices and list pointer into the indices,

Table 1: Mapper Coupler Timings for Unstructured Euler Solver (iPSC/860)

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

and

- the size of the list of indices.

and returns

- the new mapping in CSR format,
- the size of the remapped list of indices,
- a *schedule* for remapping arrays in CSR format, and
- another *schedule* for conformal data mapping.

5 Performance of Mapper Coupler Primitives

The primitives described in Section 3.3 have been implemented and have been employed in a 3-D unstructured mesh Euler solver and in a conjugate gradient structures solver. In both cases, the cost of generating the **RDG** is small compared to either the overall cost of computation or the cost of our parallelized partitioner. For instance, in our 3-D Euler solver, using a 53K mesh on 32

Table 2: Mapper Coupler Timings for Conjugate Gradient Solver(iPSC/860)

Number of Points	(Secs.)	Number of Processors					
		2	4	8	16	32	64
0.5K	graph generation	0.1300	0.0069	0.0036	-	-	-
	mapper	1.50	2.16	3.51	-	-	-
	comp/iter	0.008	0.007	0.006	-	-	-
16K	graph generation	-	-	-	1.04	0.47	0.34
	mapper	-	-	-	128.65	106.41	198.21
	comp/iter	-	-	-	0.053	0.035	0.039

processors of an iPSC/860, the execution time was 610 seconds with 100 iterations required to solve the problem. The time to generate the **RDG** using *eliminate_dup_edges* and *generate_RDG* was 1.5 seconds. For our mapper, we employed a parallelized version of Simon's eigenvalue partitioner [16]. We partitioned the **RDG** into a number of subgraphs equal to the number of processors employed. The time to of carry out the problem partitioning was 544.8 seconds; The cost of the partitioner was relatively high both because of the partitioner's high operation count and because only a modest effort was made to produce an efficient parallel implementation. The time required to generate and partition loop iterations (using *deref_rig* and *iter_partition* from Section3.3) was 3.3 seconds, this is approximately half of the cost of a single iteration of the 3-D unstructured Euler code.

6 Conclusions

We have described how to design distributed memory compilers capable of carrying out dynamic workload and data partitioning. The runtime support required for these methods has been implemented in the form of PARTI primitives. We implemented a full unstructured mesh computational fluid dynamics code and a conjugate gradient code by embedding our runtime support by hand and have presented our performance results. Our performance results demonstrate that the costs incurred by the mapper coupling primitives are roughly on the order of the cost of a single iteration of our unstructured mesh code and were small compared to the cost of the partitioner.

Acknowledgements

The authors would like to thank Geoffrey Fox for many enlightening discussions about universally applicable partitioners and Ken Kennedy for feedback on Fortran D support of compiler-linked runtime partitioning. The authors would also like to thank Horst Simon for the use of his unstructured mesh partitioning software. This work was supported by National Aeronautics and Space Administration under NASA contract NAS1-18605 while the authors were in residence at ICASE, NASA Langley Research Center. Additional support for Ponnusamy, Koelbel, and Choudhary was provided by DARPA under DARPA contract DABT63-91-C0028.

References

[1] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

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

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

[4] 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.

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

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

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

[8] M. Gupta and P. Banerjee. Automatic data partitioning on distributed memory multiprocessors. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[9] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the Fortran D programming system. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[10] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

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

[12] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

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

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

[15] 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.

[16] 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.* Permagon Press, 1991.

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

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