A Runtime Data Mapping Scheme for Irregular Problems

Ravi Ponnusamy Joel Saltz
Charles Koelbel Raja Das
Alok Choudhary

CRPC-TR92263
April 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

In Proceedings of Scalable High Performance Computing Conference, Williamsburg, VA, April 27-29, 1992, published by IEEE Computer Society Press.

SCCS - 356

A Runtime Data Mapping Scheme for Irregular Problems

by
Ponnusamy, R., Saltz, UJ., Das, R., Koelbel, C., and Choudhary, A.

in proceedings of
Scalable High Performance Computing Conference
Williamsburg, VA, April 27-29, 1992

Publisher
IEEE Computer Society Press
10662 Los Vaqueros Circle, P. O. Box 3014, Los Alamitos, CA, 90720-1264

April 1992

Syracuse Center for Computational Science
Syracuse University
111 College Place
Syracuse, New York 13244-4100
<sccs@npac.syr.edu>
(315) 443-1723

5rs356

A Runtime Data Mapping Scheme for Irregular Problems *

Ravi Ponnusamy[†], Joel Saltz[‡], Raja Das[‡], Charles Koelbel[§], Alok Choudhary[†]

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

[†]NPAC and School of Computer Science, Syracuse University , Syracuse, NY 13244

[§]Department of Computer Science, 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, a good review of this literature appears in [7]. This research has produced a variety of useful techniques, but it has typically assumed that the programmer specifies the distribution of large data structures among processor memories. Several projects have attempted to automatically derive data distributions for regular problems, (e.g. [9, 8, 1].) In this paper, we study the more challenging problem of automatically choosing data distributions for irregular problems. This work is closely related to schemes we proposed in [11] and shares important features with the distributed memory runtime parallelization schemes proposed in [10].

By irregular problems, we mean programs where the data access pattern cannot be determined during compilation. For example, the loop

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

sweeps over the edges of an unstructured mesh. This is a simplified version of a type of loop that commonly occurs 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**) is not available until runtime, this partitioning must be done on the fly. Thus, we focus on runtime mappings in this paper.

Several general heuristics have been proposed to efficiently map irregular scientific problems onto distributed memory multicomputers, some of these are described in [2, 5, 6, 14]. The codes that implement these mapping heuristics typically must be manually coupled to application programs. In this paper we describe a method by which data arrays can be automatically mapped at runtime. The mapping is based on the computational patterns in one or more user specified loops. A distributed memory compiler generates code that, at runtime, generates a distributed data structure to represent the computational pattern of the chosen loop. This computational pattern is used to determine how data arrays are to be partitioned. The compiler generates code to pass the distributed data structure to a partitioner. The work described here is being pursued in the context of the CRPC Fortran D project [7].

2 Compiler Embedded Runtime Mapping

In the scheme we present, a user labels a loop that is used to determine how a set of arrays are to be partitioned. The user also designates specific distributed arrays on which attention should be focused. The compiler generates code that, at runtime, produces a distributed data structure called the *Runtime Dependence Graph* or **RDG**. The **RDG** represents the loop's computational pattern. The compiler also generates code that passes the **RDG** to a parallelized partitioner. This partitioner uses the **RDG** to determine how to partition data. We limit ourselves to

*To appear in Proceedings of Scalable High Performance Computing Conference, Williamsburg, VA, April 27-29, 1992 array partitioning based on loops in which all designated distributed arrays conform in size and are to be identically distributed.

We generate an **RDG** using references to designated arrays in each statement of a labelled program loop. The **RDG** is constructed by adding *dataflow edge* $< i,j >$ between nodes i and j either 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, or
>
> a reference to array index j appears on the left side of a statement and a reference to i appears on the right side.

Each time dataflow edge $< i,j >$ is encountered, we increment a counter associated with $< i,j >$. Accumulation type output dependency dataflow edges of type $< i,i >$ are ignored in the graph generation process as the presence of such dependencies do not induce inter-processor communication. The **RDG** is currently represented by a distributed data structure [4], this data structure is closely related to Saad's Compressed Sparse Row (CSR) format (see [12]).

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

2.1 The Runtime Iteration Graph

We first partition distributed arrays and then, based on distributed array partitionings, partition loop iterations. In order to partition loop iterations based on a given data partition, we generate a distributed data structure we call the *runtime iteration graph* or **RIG**. The **RIG** associates with each loop iteration i, all indices of each distributed array accessed during iteration i. A **RIG** can be generated for every loop that references at least one irregularly distributed array. The *runtime iteration processor assignment graph*, or **RIPA**, is derived from the **RIG**. The **RIPA** lists, for each loop iteration, the number of distinct references to data stored on each processor.

Just as there are many possible strategies that can be used to partition data, there are also many strategies that could be used to partition loop iterations. We currently employ strategies that consult the **RIPA** to assign each loop iteration i to the processor which stores the largest percentage of the data accessed by iteration i.

2.2 Compiler-linked Mapping: Runtime Support

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

We begin with a preprocessing phase where we have an initial distribution of loop iterations. The object of this preprocessing is to extract information needed for mapping. In many cases, the initial distribution of loop iterations, I_{init}, will be a simple default distribution. In some situations (e.g. adaptive codes), preprocessing to support irregular array mappings may have already been carried out. Our runtime support will handle either regular or irregular initial loop iteration distributions I_{init}.

The preprocessing is carried out using the following mapper coupling procedures. Procedure *eliminate_dup_edges* uses a hash table to store unique dataflow edges, along with a count of the number of times each edge has been encountered. 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 dataflow edges in a loop have been recorded, *edges_to_RDG* generates the local loop **RDG** and then merges all local loop **RDG** graphs to form the loop **RDG**. The data structures that describe the loop **RDG** graph are passed to a data partitioner *RDG_partitioner*. *RDG_partitioner* returns a pointer to a distributed translation table [13], [4] 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.* Once the partitioner identifies 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 . Remap returns a pointer to a data structure which stores the communication pattern that can be used to remap the data [4].

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*.

2.3 Compiler-linked Mapping: Compiler Support

In Fortran D, a user declares a template called a *distribution* 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 explicitly specify how a distribution is to be mapped onto processors. A specific array is associated with a distribution using the Fortran D statement *align*.

In [3] we present new Fortran D syntax which implicitly specifies 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 specify whether the "owner computes" rule is to be employed or whether all work pertaining to each loop iteration is to be assigned to a single processor. [7]. We are also developing new syntax that will make it possible for a user to specify what method is be used to partition loop iterations.

The primitives described in Section 2.2 have been implemented and have been employed in a 3-D unstructured mesh Euler solver. The performance of the primitives for the Euler solver is shown in table 1. The cost of generating the **RDG** is small compared to either the overall cost of computation or the cost of our parallelized partitioner. For our mapper, we employed a parallelized version of Simon's eigenvalue partitioner [14]. We partitioned the **RDG** into a number of subgraphs equal to the number of processors employed. 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 Section2.2) is approximately half of the cost of a single iteration of the 3-D unstructured Euler code.

3 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.

Acknowledgement

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. Additional support for Ponnusamy and Choudhary have been 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] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed memory compiler methods for irregular problems - data copy reuse and runtime partitioning. In *Compilers and Runtime Software for Scalable Multiprocessors, J. Saltz and P. Mehrotra Editors*, Amsterdam, The Netherlands, To appear 1991. Elsevier.

[4] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

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

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

[14] 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.