Embedding Data Mappers with Distributed Memory Machine-Compilers

Ravi Ponnusamy *Joel Saltz*
Raja Das *Charles Koelbel*
Alok Choudhary

CRPC-TR92285
May 1992

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

Embedding Data Mappers with Distributed Memory Machine Compilers

Ravi Ponnusamy* Joel Saltz† Raja Das† Charles Koelbel‡ Alok Choudhary§

May 14, 1992

1 Introduction

In scalable multiprocessor systems, high performance demands careful attention to load balancing and communication optimization. Compilation techniques for achieving these goals have been explored extensively in recent years [3, 8, 10, 12, 15, 16] producing a variety of useful techniques. All of the above systems, however, require the user to specify the distribution of data among processor memories. A few projects have attempted to automatically derive data distributions for regular problems [11, 9, 7, 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 irregular mesh, a common operation in computational fluid dynamics. 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.

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, many heuristics have been proposed for efficient data mapping [2, 4, 5, 6, 14], but currently these partitioners must be coupled to user programs manually. In this paper we describe an automatic method for linking these partitioners. The work described here is being pursued in the context of the CRPC Fortran D project [8].

2 Compiler Embedded Run-time Mapping

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.

*NPAC and School of Computer Science, Syracuse University , Syracuse, NY 13244
†ICASE, MS 132C, NASA Langley Research Center, Hampton, VA 23666
‡Center for Research on Parallel Computation, Rice University, Houston. TX 77251
§Dept. of Electrical and Computer Engineering, Syracuse University , Syracuse, NY 13244

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, directed graph, or sparse matrix structure. 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 perform the computation.

To make our implementation tractable, we assume that 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. We also assume that all computations in a statement are executed on one processor.

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 and communications they are likely to require.

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, modified versions of the marked loops are generated to produce the required information at run time. This technique was previously used by the Kali [10] and PARTI [13] 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. 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; 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. The first-stage partitioner will divide this graph to minimize inter-processor links while while balancing the memory usage.

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

[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.

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

Number of Vertices		Number of Processors					
		2	4	8	16	32	64
3.6K	graph generation (secs.)	0.58	0.40	0.32	0.27	-	-
	mapper (secs)	15.92	11.50	12.11	14.92	-	-
	iter partitioner (secs)	0.94	0.57	0.42	0.34	-	-
	comp/iteration (secs)	2.4	1.31	0.6	0.34	-	-
9.4K	graph generation (secs.)	-	0.94	0.73	0.55	0.40	-
	mapper (secs)	-	70.96	62.3	65.2	89.7	-
	iter partitioner (secs)	-	1.19	0.82	0.60	0.43	-
	comp/iteration (secs)	-	4.83	2.35	1.1	0.67	-
54K	graph generation (secs.)	-	-	-	-	1.92	1.28
	mapper (secs)	-	-	-	-	544.81	673.14
	iter partitioner (secs)	-	-	-	-	3.30	3.03
	comp/iteration(secs)	-	-	-	-	6.06	3.81

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.

3 Performance of Mapper Coupler Primitives

A set of primitives implementing the ideas in Section 2 have been implemented and have been employed in a 3-D unstructured mesh Euler solver. In that application, the cost of generating the **RDG** is small compared to either the overall cost of computation or the cost of our parallelized partitioner, as shown in Table 1. Graph generation (including both the **RDG** and **RIG** is approximately the same cost as a single computatihonal iteration; approximately 100 iterations are needed to solve the full problem. We used a parallelized version of Simon's eigenvalue partitioner [14] for the data partitioning. 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. We have also tested our methods on a conjugate gradient code, obtaining similar results.

4 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] 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] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[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.

[15] 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.

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