Parallel Implementations of Irregular Problems Using High-Level Actor Language

R. B. Panwar*
Application Dev. Technology Institute
IBM Santa Teresa Labs
San Jose, CA 95141, USA
Email: panwar@vnet.ibm.com

W. Kim and G. A. Agha
Open Systems Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: { wooyoung | agha}@cs.uiuc.edu

Abstract

In this paper we present our experience in implementing several irregular problems using a high-level actor language. The problems studied require dynamic computation of object placement and may result in load imbalance as the computation proceeds, thereby requiring dynamic load balancing. The algorithms are expressed as fine-grained computations providing maximal flexibility in adapting the computation load to arbitrary parallel architectures. Such an algorithm may be composed with different partitioning and distribution strategies (PDS's) to result in different performance characteristics. The PDS's are implemented for specific data structures or algorithms and are reusable for different parallel algorithms. We demonstrate how our methodology provides portability of algorithm specification, reusability and ease of expressibility.

1 Introduction

In parallel computing a set of operations and the partial order in which they may be carried out define an *ideal algorithm* [7]. The ideal algorithm may be specified without introducing any unnecessary sequentiality by using maximally concurrent objects, i.e., *actors*. In practice, limitations on computation and communication resources in practical architectures make implementations of a parallel algorithm use only part of all available parallelism in the ideal version of the algorithm. In particular, how the computation and the data are placed determines which potentially parallel operations in an algorithm may be executed sequentially. As a result, different placement policies lead to different performance.

In this paper we study the impact of computation and data placement on the execution efficiency of irregular problems, i.e., problems where the placement of subcomputations and their communication topologies are dynamic. We adopt a programming methodology which uses fine-grained computation, asynchronous communication and dynamic object creation [14]. Such a methodology enables programmers to compose different partitioning and distribution strategies (PDS's) with ideal algorithms for better performance. A PDS which is defined for a particular data structure or an algorithm may be reused with different parallel algorithms. We have implemented several irregular problems with different communication and computation characteristics and composed them with different partitioning strategies. These implementations have been evaluated through measurement taken on a CM-5. We present the evaluation results as well as what we have learned from the implementations.

2 Background

Most of the previous work in specifying and optimizing placement of data in parallel programs has been based on extensions of sequential languages. In particular, Fortran-D [5] and High Performance Fortran (HPF) [11] allow explicit specification of data decomposition and distribution policies for regular problems to improve execution efficiency on distributed memory multicomputers. For irregular problems, such *a priori* determination of the necessary data distribution is not feasible. To address this problem in some limited cases, PARTI [4] and Kali [10] transform a user-defined `for` loop to an *inspector/executor* pair. In these languages the compiler assumes the entire responsibility to uncover concurrency characteristics. In many regular problems using dense matrices, compiler tools may exploit most of the useful parallelism. However, an unaided compiler may be less successful in more general cases.

Because specifying a parallel algorithm in terms of actors does not introduce unnecessary sequentiality we use Actors as our computational model of concurrency [1]. Actors encapsulate data, procedures and a thread of control. Each actor has a *mail address* and a *behavior*. Mail addresses

*The first author was at the Open Systems Laboratory, University of Illinois Urbana-Champaign when the work related to the paper was done.

1063-7133/96 $5.00 © 1996 IEEE
Proceedings of IPPS '96

may be communicated, thereby providing a dynamic communication topology. In response to a communication, an actor may: (*i*) asynchronously send a message to a specified actor, (*ii*) create an actor with the specified behavior, and (*iii*) change its local state.

These basic actor primitives are supported as primitives in THAL. THAL is a descendent of HAL [6, 2, 3] and *tailored* for high-performance execution on stock-hardware distributed memory multicomputers, such as the CM-5. It extends `create` primitive to take an additional location argument to provide programmers with control over actor placement. In addition, migration for dynamic actor relocation is supported at the language level. A program written using the high-level actor abstraction is translated into a set of C programs. Since a C compiler may optimize the sequential intra-method computation, the compile-time optimizations of the THAL compiler concentrate on improving concurrent inter-method execution. In particular, the THAL compiler restores profitable concurrency that may be lost by some high-level language constructs [2]. Additional efficiency is supported by the runtime kernel [9] which provides a fast communication layer implemented using the CMAM [15]. Despite the flexibility and support for dynamic computations, its performance on dense, regular problems is competitive with more restrictive, static languages [9].

3 Methodology

Fine-grained specification of parallel algorithms provides maximal flexibility in distributing the workload. THAL programs may exploit this flexibility to compose a parallel algorithm with different PDS's [14]. The PDS's themselves are designed for specific data structures or program structures. For example, more than one PDS's may be designed for a binary tree data structure. There are a number of algorithms which explicitly or implicitly construct the binary tree data structure and these algorithms may be composed with such PDS's. Varying the PDS composed with an algorithm is often critical in assuring optimal performance. First, the differences in the communication and computation requirements of different architectures make one PDS better suited than another. Second, the choice of a PDS may be affected by the input. For example, a PDS for a parallel algorithm that results in a very unbalanced binary tree may be different from that for an algorithm generating a balanced binary tree. Figure 1 shows how the separate specification of PDS and ideal algorithm may be combined. The glue code specifies details such as which algorithm behavior matches with which PDS behavior. This separate specification promotes modularity, portability and reusability [14]. Figure 2 gives the block diagram of the system used for our experiments. An ideal algorithm composed with a PDS using some glue code is translated to a single THAL program. The program is compiled to the code for the actor run-time kernel and executed on a CM-5. Below, we present details of our experiments and the measurements taken on a CM-5.

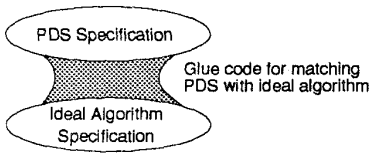

Figure 1. Composing a PDS with an ideal algorithm Specification

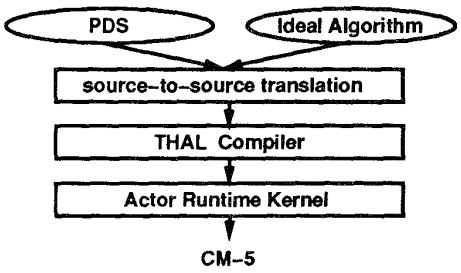

Figure 2. A Block diagram of the system.

4 Irregular Problems Based on Tree Structure

There are a number of tree-based problems including problems solved using divide and conquer strategy and problems based on a search tree. In our experiment we used a problem generating a binary tree structure with the following PDS's:

Subtree-subcube (SS) PDS assumes the given architecture can be recursively divided into subcubes; it recursively divides the binary tree into subtrees and maps the subtrees onto subcubes. Specifically, if the number of processors P available to be a power of two, the subtree-subcube PDS divides the given tree into P subtrees and maps a complete subtree to each processor. This PDS results in low communication overhead but may cause load imbalance if the load depends on the size of a subtree and if the sizes of the subtrees mapped to different processors are significantly different.

Subtree-subcube-lb (SS-lb) PDS divides the tree structure into n small subtrees where n is significantly larger than the number of processors. These subtrees are uniformly allocated to the processors. Since the chances of one processor getting a large subtree are reduced,

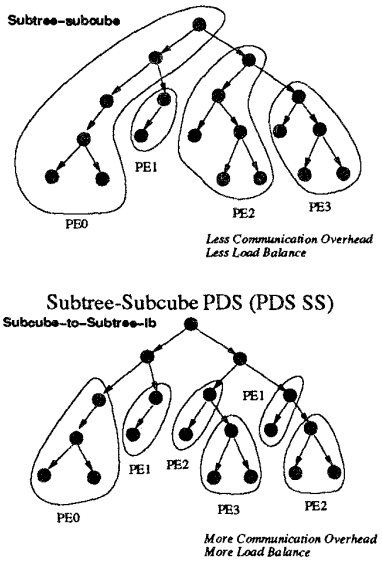

Figure 3. PDS's for the Binary Tree Structure

this strategy results in better load balance characteristics but higher communication overhead than the subtree-subcube PDS described above.

The algorithm we implemented computes the sum of elements available at each node of the tree (similar to the summation of leaves of a binary tree described in [12]). The values being summed are arrays of floating point numbers. The tree is generated as a search tree formed by inserting nodes containing a key. By assigning random values with different distributions to the keys, it is possible to change the structure of the tree in a controlled fashion thereby allowing changes to the load distribution. In our implementation, the keys were selected from a range [0:10000] and the root node was always given the key 5000. If the keys are uniformly distributed in the range [0:10000] the tree is expected to be approximately balanced. On the other hand, if the keys are distributed so that 66% of the values fall in the range [0:5000] and the remaining fall in the range [5000:10000] an unbalanced tree is generated with approximately twice the number of nodes in the left subtree as in the right subtree.

Results were obtained for a binary tree of 256 nodes and 1024 nodes, performing the summation where each node stored an array of 400 elements. Figure 4 gives the speedup obtained when the tree is approximately balanced. Note that for the small number of nodes (256 nodes) the imbalance caused by the randomness of the subtrees is significant and therefore the subtree-subcube-lb PDS performs better in most of the range. For larger trees (1024 nodes) the effect of

PEs	*Time 256 Nodes*		*Time 1024 Nodes*	
	SS	*SS-lb*	*SS*	*SS-lb*
1	0.159	0.159	0.638	0.634
2	0.080	0.099	0.321	0.444
4	0.073	0.060	0.257	0.277
8	0.054	0.037	0.217	0.289
16	0.030	0.032	0.196	0.234
32	0.023	0.023	0.143	0.103

Table 1. Timing results (in seconds) for Tree Summation (balanced tree).

PEs	*Time 256 nodes*		*Time 1024 nodes*	
	SS	*SS-lb*	*SS*	*SS-lb*
1	0.159	0.157	0.639	0.634
2	0.111	0.091	0.446	0.429
4	0.085	0.070	0.396	0.246
8	0.062	0.042	0.330	0.225
16	0.056	0.034	0.184	0.159
32	0.046	0.025	0.161	0.106

Table 2. Timing results (in seconds) for Tree Summation (unbalanced tree).

the randomness of the tree is minimized resulting in a better balance in the size of the subtrees. As a result, the overhead caused by the subtree-subcube-lb PDS does not offset the load balance offered. On the other hand, the Figure 5 gives the speedup obtained when the binary tree is unbalanced such that the the size of the subtree to the left of the root node is 70% of the total subtree. As a result, for both small (256 nodes) and large (1024 nodes) trees the performance of the subtree-subcube-lb PDS is better than the subtree-subcube PDS irrespective of the machine size. It can also be seen that as the number of processors increases the relative performance of the subtree-subcube-lb PDS improves as compared to subtree-subcube PDS.

5 Irregular Problems based on Master Worker Structure

In the master worker configuration a master actor distributes the work among several worker actors. The workers may interact with the master or simply return the results when they are done. Performance results for two problems based on such a structure are discussed below, i.e., adaptive quadrature and the grid problem.

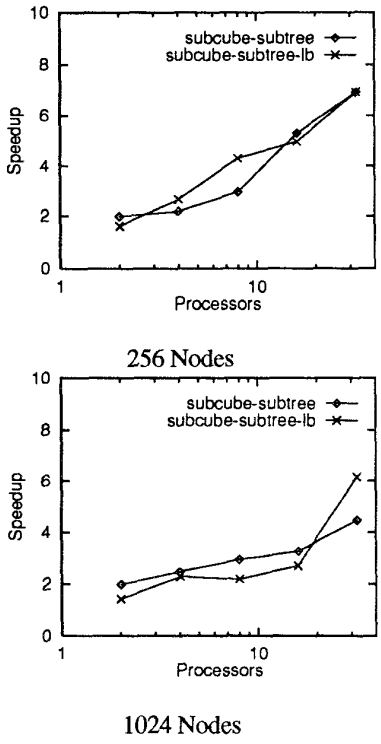

Figure 4. Speedup of Tree Summation (balanced tree)

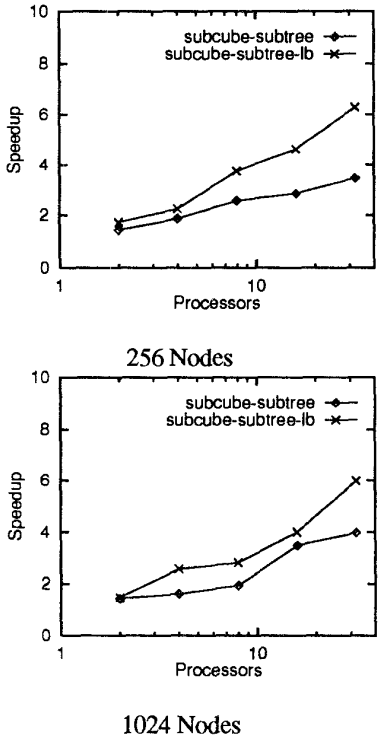

Figure 5. Speedup of Tree Summation (unbalanced tree)

5.1 Adaptive Quadrature

Integration algorithms can be implemented in parallel using a master worker configuration such that the given interval is divided into subintervals with each interval passed to a worker. The workers perform the integration in their subintervals in parallel and return the values to the master where they are combined to obtained the final result. Adaptive integration techniques [8] vary the step size used for the integration in a region based on the local error estimate. Thus the work available for each worker may dynamically increase as the computation proceeds. We allow the workers to create new workers, thereby dynamically changing the number of workers and keeping constant the amount of work allocated to each worker. A worker that decides to decrease its step size by a constant factor (assumed an integer K) divides its local interval by the factor K and creates new workers to handle the extra subintervals.

In the experiment the following function is integrated in the interval from 0 to 10π.

$$f(x) = \begin{cases} c_1 & 0 \le x \le 5\pi \\ \mid c_2 \sin(c_3 x) + c_1 \mid & 5\pi < x \le 10\pi \end{cases}$$

Note that the function is such that it requires a coarse grid size for integration in some subintervals and a much finer grid size in other subintervals. The value of all the constants c_i is equal to 1000. The initial grid size is 0.001 and the error bound is 10^{-5}. The number of worker actors created was 32. The interval was initially divided into 32 subintervals each given to a worker actor. The total number of workers created dynamically during the computation was 10.

There are two different partitioning strategies involved in this implementation: a static partitioning strategy that governs the placement of initial workers and a dynamic partitioning strategy that decides the placement of new workers to be created as the load changes. The performance results of an implementation that uses only the static placement strategy is compared with an implementation that also allows dynamic placement. The dynamic placement of all such new workers is determined by a centralized actor. It may seem that such a centralized placement strategy may result in a bottleneck but actual implementation results show an improvement in performance for the problem and architecture sizes considered (see Table 3 and Figure 6). For all machine sizes considered, the performance obtained with DLB is better than that obtained without DLB.

PEs	*Time Without DLB*	*Time With DLB*
1	0.963	0.963
2	0.867	0.867
4	0.435	0.386
8	0.211	0.169
16	0.146	0.122
32	0.073	0.051

Table 3. Timing results (in seconds) for Adaptive Quadrature Algorithm.

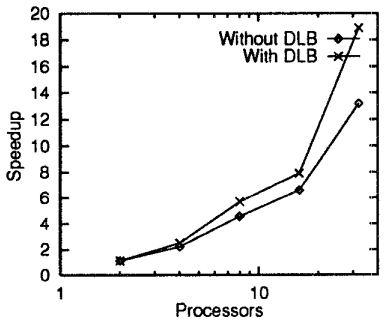

Figure 6. Speedup of Parallel Adaptive Quadrature Implementation

5.2 Unstructured Grid Problem

The unstructured grid problem [12, 13] solves a set of differential equations for a given input domain with adaptive refinement of the grid used for solving the equations. The problem can be modeled using the same master worker configuration used in the adaptive quadrature problem. The differences between the structures of the two problems are that the workers in the unstructured grid problem communicate with each other and that the values computed for each grid point are stored and reused in further iterations. Since the workers communicate with each other, their relative placement may affect performance. As with the adaptive quadrature algorithm, the grid size is refined in certain regions of the domain where the error is high. Hence, two placement strategies are combined to provide good performance: a static placement strategy which decides the initial placement of the workers and a dynamic placement strategy which decides the placement of the workers created dynamically as the computation proceeds.

We created 32 workers in the THAL implementation of the irregular grid solution of the heat equation in one dimension. The initial grid was divided into 32 equal parts and given to one actor each. Initially each actor processed 512 grid points. As the computation proceeds two of the workers refine their grid, dividing their interval into four subintervals and reducing their grid size by a factor of four. They create four new workers each for processing the new workload created. The placement of these new workers is decided randomly. The timing results show that even with a random dynamic placement strategy the performance is better than that obtained from a strategy that does not dynamically modulate the load imbalance created as the computation proceeds (see Table 4 and Figure 7).

PEs	*Time Without DLB*	*Time With DLB*
1	1.117	1.118
2	0.584	0.593
4	0.343	0.341
8	0.231	0.189
16	0.181	0.131
32	0.154	0.093

Table 4. Timing results (in seconds) for Irregular Grid Problem

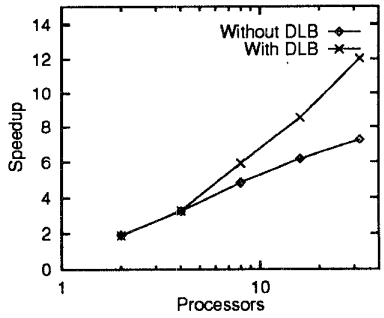

Figure 7. Speedup of Parallel Implementation of Irregular Grid Problem

6 Discussion

In this paper we have presented our implementation methodology for irregular problems and the performance results of three irregular applications with different PDS's. The separate specification of a PDS and an ideal algorithm promotes modularity and allowed us to reuse the algorithm

specification throughout the experiments. By using fine-grained computation i.e., *actors* in the algorithm specification, we were able to naturally express the useful concurrency characteristics existing in the problems. All the problems implemented had load that changes with the program execution. The results showed that the use of dynamic load balancing improved performance.

Acknowledgments

The research has been made possible by support from the Office of Naval Research (ONR contract number N00014-93-1-0273), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195 and NSF CCR-9312495).

We would like to acknowledge Dan Sturman for his careful review of the draft of this paper. We also thank the UIUC NCSA for use of their CM-5 machine.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] G. Agha, C. Houck, and R. Panwar. Distributed Execution of Actor Systems. In D. Gelernter, T. Gross, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 1–17. Springer-Verlag, 1992. LNCS 589.

[3] G. Agha, W. Kim, and R. Panwar. Actor languages for specification of parallel computations. In G. E. Blelloch, K. Mani Chandy, and S. Jagannathan, editors, *DIMACS. Series in Discrete Mathematics and Theoretical Computer Science. vol 18. Specification of Parallel Algorithms*, pages 239–258. American Mathematical Society, 1994.

[4] R. Das, R. Ponnusamy, J. Saltz, and D. Mavriplis. Distributed Memory Compiler Methods for Irregular Problems - Data Copy Reuse and Runtime Partitioning. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*. Elsevier Science Publishers, 1992.

[5] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran-D for MIMD Distributed Memory Machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[6] C. Houck and G. Agha. HAL: A high-level actor language and its distributed implementation. In *Proceedings of the 21st International Conference on Parallel Processing (ICPP '92)*, volume II, pages 158–165, St. Charles, IL, August 1992.

[7] L. H. Jamieson. Characterizing parallel algorithms. In R. J. Douglass L.H. Jamieson, D.B. Gannon, editor, *The Characteristics of Parallel Algorithms*, pages 65–100. MIT Press, 1987.

[8] D. Kahaner, C. Moler, and S. Nash. *Numerical Methods and Software*. Prentice Hall, 1989.

[9] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Supercomputing '95*, 1995.

[10] C. Koelbel and P. Mehrotra. Compiling Global Name-space Parallel loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, 1991.

[11] David. B. Loveman. High Performance Fortran. *Parallel & Distributed Technology, Systems & Applications*, 1(1):25–42, February 1993.

[12] K. Mani Chandy and Stephen Taylor. *An Introduction to Parallel Programming*. Jones and Bartlett Publishers, Boston, 1992.

[13] P. Mehrotra, J. Saltz, and R. Voigt, editors. *Unstructured Scientific Computation on Scalable Multiprocessors*. MIT Press, 1992.

[14] R. Panwar and G. Agha. A Methodology for Programming Scalable Architectures. *Journal of Parallel and Distributed Computing*, 22(3):479–487, September 1994.

[15] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of International Symposium of Computer Architectures*, pages 256–266, 1992.