Parallel Programming Support for Unstructured Mesh Computation

Jyh-Ching Hong Pangfeng Liu Tzu-Hao Shen Jin-Xian Wu
Chih-Hsuae Yang
Department of Computer Science and Information Engineering
National Chung Cheng University
Chiayi, Taiwan 621, R.O.C.

Abstract *This paper describes the implementation of a platform-independent parallel unstructured mesh library. Unstructured mesh is a fundamental data structure in many irregular scientific computations. However, it is very difficult to implement an unstructured mesh efficiently on parallel computers because of the complex programming details in handling irregular distribution of mesh points. To overcome this problem, we developed an object-oriented framework in which it is easy to implement and maintain unstructured meshes. We demonstrated the versatility of this framework by implementing a flux simulation using roe scheme and an electro-magnetic simulation on both distributed and shared memory parallel computers, using the same application programming interface provided by our framework. The timing results on these different parallel platforms, including PC/workstation clusters and shared-memory multiprocessors, are also reported.*

Keywords: Unstructured mesh, parallel processing, object-oriented programming

1 Introduction

Unstructured mesh is a fundamental data structure in many irregular scientific computations. These applications discretize the computation domain into a set of mesh points that each has physical quantities of interest. The arrangement of mesh points is completely determined by the application and in general will not exhibit any regular pattern. Therefore the mesh point distribution can be highly irregular and dynamically adaptive to the evolution of the ongoing computation. The mesh points are connected by edges that represent the interaction among them, which is determined by the governing equation of the physic system.

1.1 The unstructured mesh applications

Unstructured meshes are widely used in problems where the data or the computation structure is non-uniform. For example, an unstructured mesh can be used to model the resonance of piezoelectric crystals [3], the surface of aerospace vehicles, the electro-magnetic wave passing through a material [14], or to simulate the vortices in the superconductors while they arrange themselves in a hexagonal lattice pattern in order to minimize the free energy when the temperature is below a critical point [8].

1.2 Object-oriented support for unstructured mesh

Although different unstructured mesh computations perform different calculations according to different governing equations, they use the mesh virtually in the same way. The applications traverse all the mesh points and process the data stored within. A mesh point updates its stored data, which represent the physic characteristics at that point, by retrieving the data from its neighbors and performing calculations on them. The mesh applications

may have different calculation rules and communication patterns, but the principle of exchanging data with neighbors to update one's own data remains the same.

Our library tries to extract the common ingredients from different unstructured mesh computations and reuse them systematically. We divide an unstructured mesh computation into two logical levels: the library and the application, then set up a clear interface between them so that tedious details of maintaining an unstructured mesh can be hidden from the application programmers. The application programmers only have to concentrate on the application-dependent computation kernel, and let the library handle the details.

We implemented this interface between the unstructured mesh library and the application with *object-oriented technology*. We implemented various object classes within the library to capture the fundamental properties of an unstructured mesh. Users of the library can inherit the generic unstructured mesh, customize it by adding application-dependent data, and redefine generic methods inherited from the library to suit their needs.

One immediate advantage of using objects is that the library can be easily ported to a parallel environment. By specifying a fixed interface between the application and the library objects, the application code is completely independent from how and where the library will be implemented. If we want to port a scientific computing application from one platform to another, we only have to modify the implementation of the communication object in our library — the users do not need to change any part of their programs. In addition, we are free to choose any implementation to maximize the efficiency for a particular computer, be it a PC cluster or a large-scale parallel computer.

1.3 Related Works

The benefit of data abstraction in object-oriented languages on scientific code development has been demonstrated by various efforts. For example, C++ objects are used to define data structures with built-in data distribution capabilities. Examples of work along this line include the Paragon package [4], which supports a special class PARRAY for parallel programming, the A++/P++ Array class library [15], PC++ proposed by Lee and Gannon [12, 18], which consists of a set of distributed data structures (arrays, priority queues, lists, etc.) implemented as library routines, where data are automatically distributed based on directives. Interwork II Toolkit [2] described by Bain supports user programs with a logical name space on machines like iPSC. The user is responsible for supplying procedures mapping the object name space to processors. Unfortunately, all of these efforts use static arrays and will have difficulties in representing dynamic structures efficiently. The current implementation of our library uses a dynamic pointer-based structure, which is the most intuitive and convenient way to handle the adaptive nature of unstructured mesh.

Our effort has similar goals and approaches to the POOMA package [1] and the Chaos++ library [16]. POOMA supports a set of distributed data structures (fields, matrices, particles) for scientific simulations. To our knowledge, POOMA has not supported adaptive data structures as our library does. Chaos++ is a general-purpose runtime library that supports pointer-based dynamic data structures through an inspector-executor-based runtime preprocessing technique. On the other hand, our framework focuses on unstructured mesh and is able to exploit optimizations that would be difficult for a general preprocessing technique to find.

In addition to the above work on object-oriented parallelism which has influenced ours, a large body of work in the literature can be categorized as "object-parallelism," where *objects* are mapped to *processes* that are driven by *messages*. If a message is sent in between two processes residing on two different processors, this message will be implemented via inter-processor communication. Examples of parallel C++ projects using this paradigm include the Mentat Run-time Sys-

tem [9], Concurrent Aggregates (CA) [5] by Dally et al., and VDOM by [7]. Our use of object-orientation is for structuring the unstructured mesh and their specializations for optimizations, debugging, profiling, etc., which is entirely distinct in philosophy from that of object-parallelism. Applying these ideas on dynamic tree structures, we reported abstractions of adaptive load balancing mechanisms and complex, many-to-many communications as C++ classes for supporting tree-based scientific computations [13].

2 Data partitioning

2.1 Distributed memory implementation

In the distributed memory version of our library the mesh is partitioned into sub-meshes and distributed among local memories of processors. In order to obtain the same computation results as in the global view, the local computations must be coordinated. We adopt the owner-computes rule, which distributes computations according to the mapping of data across processors. However, a local sub-mesh may require data from other processors to complete the computation assigned to it. When communications mostly occur between neighboring processors and the same communication patterns may occur many times during program execution, it is more efficient to duplicate boundary data elements on adjacent processors.

We classify the data into two categories, *master* copy and *duplication*. A master copy is a data region in the original global structure that is mapped to a processor. A master copy can make copies of itself, called duplication, on other processors. As far as each master copy is concerned, there is no distinction between global and local structures. The computations read and update the master copy only – the duplications only provide data and are read-only. Therefore, data coherence is guaranteed by allowing only the master copy to be updated, and only one master copy exists for one data element. To assure synchronization, data elements are duplicated before the actual computation is performed. After data are partitioned, system objects in the unstructured mesh layer duplicate the data to the processors where they are essential to the computation.

2.2 Shared-memory implementation

We also implemented a shared-memory version of our library. Since every processor can now access the global shared-memory, the data duplication is no longer needed. However, the implementation still partitions the mesh into sub-meshes, and each processor is still responsible for updating the data that it owns.

Since the data is no longer duplicated, the sequence of updating the mesh data on the processor boundary becomes critical. In our implementation we used a critical section to ensure that no two processors will access or update a mesh point simultaneously. We noticed that by doing so the library implementation becomes much easier since the complicated details of retrieving and refreshing the duplicated data are avoided.

3 The unstructured mesh library

We divide the library into *unstructured mesh layer* and *application layer*. The unstructured mesh layer contains basic graph operations and the necessary data structures for implementing a general directed graph, as well as special functions for manipulating unstructured mesh. The unstructured mesh layer constitutes the unstructured mesh API that *application layer* can use to develop unstructured mesh applications.

3.1 Application layer

The library users write application by inheriting classes from the unstructured mesh layer. In other words, the application layer consists of customized classes inherited from unstructured

mesh layer, which have additional application-dependent data and operations. The users must define their data in a mesh node and provide functions to operate on the data they defined.

3.1.1 Fluid dynamic in Roe Scheme

We first wrote a flux simulation program [17] using Roe's scheme. First we define the data type that contains all the necessary data for computational fluid dynamic in Roe Scheme. Then we give this user-defined data type `Fluxroe_node` to the container class `Mesh` and `Mesh_node` of our library, to form the actual data types for unstructured mesh and mesh points respectively. Then we put the flux computation kernel into `process` of the compute class `Fluxroe_compute`, which is specific to traversing a graph consists of `Fluxroe_node` data. For the flux simulation, the `process` function goes through every edge adjacent to the current node, fetches the data from the neighbors, use Roe's scheme to compute the results, and finally updates the data in the current mesh node accordingly.

3.1.2 Electro-magnetic Application: EM3D

We also wrote an electro-magnetic application EM3D [6] using our library. This application simulates an electro-magnetic wave passing through a material. The same as we implemented the flux application, we started with a data type `Em3d_data` that contains all the application-dependent data, then define the mesh for EM3D computation.

4 Experimental results

To evaluate the efficiency of our library we develop two exemplary programs and measure their execution time. The test machines include a Pentinum PC cluster connected with ethernet, a SUN UltraSparc workstations cluster connected with fast ethernet, and shared-memory machine like SUN Enterprise 5000. The major difference among these machines is the communication speed. The PC cluster has the slowest communication, and the shared-memory SUN enterprise has the highest bandwidth.

The exemplary programs we implement and test are a flux simulation program (fluxroe) [17] and an electro-magnetic simulation program (EM3D) [6]. The fluxroe is a flux simulation program used in airfoil design. We isolate the computation kernel after intensive study of the original C code, then rewrote it as a module that can be plugged into our framework. We generate the input data by using the original generator in [17]. This generator produces a random unstructured mesh in which each edge is chosen with a fixed probability [1]. The other exemplary programs is EM3D [6] described in the previously. The original version of this program was written in split-C language [6]. We translated it into C and parallelize it with our library. We also wrote an unstructured mesh generator for the EM3D program to generates the bipartite graph input for the EM3D.

4.1 Fluxroe

4.1.1 Distributed memory implementation

We first ran the fluxroe code on a cluster of 8 200MHz Pentium Pro cluster connected by ethernet. Each PC has 128 mega-bytes of physical memory running Red Hat linux 4.2. Table 1 gives the timing results from this experiment[2].

We discover two major overheads due to the parallelization. First, we introduced complex data structures and operations for data sharing and message passing, and large overhead associated with them. Secondly, the communication on ethernet is expensive. We partially solved the problem by using faster communication like fast ethernet. Table 2 gives the improved timing results from running the same code on a 4 UltraSparc workstation clus-

[1] We used 0.5 throughout the experiments.

[2] All the timing results in this section does not include the initialization stage.

Table 1: Execution time of fluxroe per iteration on a 8 PC cluster.

Execution Time Per Traverse				
mesh points	16384	36864	65536	147456
c version	0.124	0.289	0.519	1.260
c++ version	0.314	0.720	1.289	2.922
2 nodes	0.283	0.677	1.160	2.667
4 nodes	0.144	0.335	0.623	1.395
8 nodes	0.083	0.160	0.310	0.710

ter connected with fast ethernet. In addition, we have used better partitioning tools (like *Metis* [10, 11]) to reduce the communication volume.

Table 2: Execution time of fluxroe on a Sun UltraSparc workstations cluster connected with fast ethernet.

Execution time per traverse					
mesh points	36864	65536	147456	262144	589824
c version	1.076	1.770	4.168	7.156	16.490
2 nodes	0.922	1.574	3.150	5.428	188.937
4 nodes	0.380	0.711	1.920	2.949	6.329

4.1.2 Shared memory implementation

We implement the fluxroe on a shared memory SUN Ultra Enterprise with 8 CPUs and 1004M bytes of memory. The same code is tested on a dual CPU Pentium II-300 with 128M byte memory. The implementation uses Pthread to synchronize and simplify the communication among processors. Both codes are compiled with g++. The timing results are shown in Table 3.

Table 3: Execution time of fluxroe

on a Sun Ultra Enterprise			
mesh size	400	10000	40000
1 processor	3.36	103.70	452.23
2 processor	2.07	59.40	246.76
4 processor	1.21	33.04	137.61
8 processor	0.67	18.02	74.37
on a dual CPU Pentium PC			
mesh size	400	10000	40000
1 processor	2.65	92.96	402.59
2 processor	1.61	46.87	210.69

4.2 EM3D

4.2.1 Distributed memory implementation

We then implement a EM3D [6] code with our library. First we ran the code on the same Pentium Pro cluster where we ran the fluxroe and Table 4 shows the timing results. We observe that the performance is not satisfactory. When the number of processors increases to eight, the performance degrades unacceptably. We conjecture that for EM3D the communication overheads is much larger than the parallelization benefit that we can obtain on a slow network. To verify this conjecture we run the EM3D on the Sun UltraSparc cluster mentioned earlier.

Table 4: Execution time of EM3D

on a Pentium Pro PC cluster					
mesh points	9612	16384	36864	65536	147456
c version	0.100	0.189	0.489	0.833	1.966
2 nodes	0.211	0.467	1.011	1.555	3.967
4 nodes	0.201	0.344	0.544	0.833	2.133
8 nodes	0.344	0.633	0.711	1.755	3.288
on a Sun UltraSparc workstation cluster					
mesh points	16384	36864	65536	147456	262144
c version	0.096	0.211	0.372	0.866	1.472
2 nodes	0.081	0.189	0.346	0.809	1.226
4 nodes	0.064	0.103	0.170	0.350	0.709

Table 4 also gives the timing results of running the EM3D on a UltraSparc cluster. By comparing Table 4 with Table 2 we observe that the EM3D application took much less time per iteration than fluxroe did, due to the smaller number of floating point operations required for each mesh point computation. As a result EM3D has a much higher communication to computation ratio. That is, EM3D needs more time in communication relatively than fluxroe and does not have a good speedup

number, even in fast ethernet connection.

4.2.2 Shared memory implementation

The timing results from EM3d on shared memory implementation are shown in Table 5. Due to the small amount of computation per mesh node, the overall performance gain is not as good as in the fluxroe.

Table 5: Execution time of Em3d.

on a Sun Ultra Enterprise			
mesh size	400	10000	40000
1 processor	0.57	27.56	150.33
2 processor	0.74	22.24	107.34
4 processor	0.86	15.19	67.96
8 processor	1.46	11.90	48.28
on a dual CPU Pentium PC			
mesh size	400	10000	40000
1 processor	0.78	32.41	149.69
2 processor	0.58	18.15	88.57

5 Conclusion and future works

The object-orient framework makes it easy to develop unstructured mesh applications. The users only have to take care of the computation aspects and do not need to consider parallelization or mesh structure implementations. Once the unstructured mesh library is built, we can easily construct unstructured mesh applications by class inheritance and redefining those application-dependent virtual functions. By using the library, the time saved in application development can now be shifted into designing new computation models and analyzing the experimental results.

The object-orient approach also makes the user codes robust and easy to maintain. The user program will not change even when we modify the implementation of the library. For example, if we want to change the communication protocol in the library, we just rewrite those communication-related classes and do not need to modify any user codes. In addition, since user codes reuse thoroughly tested library routines, they are robust and easy to debug.

One insightful experience we learned from the implementation is that it is vital to design the objects and their interfaces very very carefully. A well-designed object not only increases the readability of the code but also makes the software components more reusable. The objects must clearly represent entities in the computation model and the user interface must be clear, simple, and intuitive. An object-oriented library can deliver all the software management advantages only when people use it.

Another observation is that before an application is to be parallelized, careful evaluation must be made to determine the possible performance gain by using parallel computers. For example, in the EM3D application, the computation does not require significant time. As a result the ratio of computation to communication prohibits any significant speedup in a loosely coupled parallel environment.

References

[1] Susan Atlas, Subhankar Benerjee, Julian C. Cummings, Paul J. Hinker, M. Srikant, John V.W. Reynders, and Marydell Tholburn. Pooma: A high performance distributed simulation environment for scientific applications. In *Supercomputing95*, 1995.

[2] W. L. Bain. Aggregate distributed objects for distributed memory parallel systems. In *The 5th Distributed Memory Computing Conference, Vol. II*, pages 1050–1055, Charleston, SC, April 1990. IEEE.

[3] Tom Canfield, Mark T. Jones, Paul E. Plassam, and Michael Tang. Thermal effects on the frequency response of piezoelectric crystals. *New Methods in Transient Analysis*, PVP-Vol. 246 and AMD-Vol. 143:103–108, 1992. ASME.

[4] C. M. Chase, A. L. Cheung, A. P. Reeves, and M. R. Smith. Paragon: A parallel programming environment for scientific applications using communication structures. In *1991 International Conference for Parallel Processing, Vol. II*, pages 211–218, August 1991.

[5] A. A. Chien and W. J. Dally. Concurrent aggregates (CA). In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–196, Seattle, Washington, March 1990. ACM.

[6] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. Technical report, Computer Science Division University of California, Berkeley.

[7] M. J. Feeley and H. M. Levy. Distributed shard memory with versioned objects. In *OOPSLA '92*, pages 247 – 262, Vancouver, BC, Canada, October 1992.

[8] Lori Freitag, Mark T. Jones, and Paul E. Plassam. New techniques for parallel simulation of high-temperature superconductors. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 726–733, Knoxville, Tennessee, 1994. IEEE.

[9] A. Grimshaw. The mentat run-time system: Support for medium grain parallel computation. In *The 5th Distributed Memory Computing Conference, Vol. II*, pages 1064–1073, Charleston, SC, April 1990. IEEE.

[10] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. Technical report, University of Minnesota, Department of Computer Science, Minneapolis, 1996.

[11] George Karypis and Vipin Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. Technical report, University of Minnesota, Department of Computer Science, Minneapolis, 1996.

[12] J. K. Lee and D. Gannon. Object oriented parallel programming experiments and results. In *Supercomputing '91*, pages 273–282, November 1991.

[13] Pangfeng Liu and Jan-Jan Wu. A framework for parallel tree-based scientific simulations. In *Proceedings of the 1997 International Conference on Parallel Processing*, 1997.

[14] N. K. Madsen. Divergence preseriving discrete surface integral methods for maxwell's curl equations using non-orthogonal unstructured grids. Technical report, Technical Report 92.04, RIACS, February 1992.

[15] R. Parsons and D. Quinlan. A++/p++ array classes for architecture independent finite difference calculations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.

[16] J. Saltz, A. Sussman, and C. Chang. Chaos++: A runtime library to support distributed dynamic data structures. *Gregory V. Wilson, Editor, Parallel Programming Using C++*, 1995.

[17] Jan-Jan Wu, Joel Saltz, Seema Hiranandani, and Harry Berryman. Runtime compilation methods for multicomputers. In *Proceedings of the 1991 International Conference on Parallel Processing*, 1991.

[18] S. X. Yang, J. K. Lee, S. P. Narayana, and D. Gannon. Programming an astrophysics application in an object-oriented parallel language. In *Scalable High Performance Computing Conference SHPCC-92*, pages 236 – 239, Williamsburg, VA, April 1992.