平行不規則結點計算之物件導向支援環境
Object-Oriented Support for Parallel Unstructured Mesh Computations

洪志青
Jyh-Ching Hong
中正大學資訊工程系
Department of Computer Science
National Chung Cheng University
hjc85@cs.ccu.edu.tw

劉邦鋒
Pangfeng Liu
中正大學資訊工程系
Department of Computer Science
National Chung Cheng University
pangfeng@cs.ccu.edu.tw

摘要

本論文描述一平台獨立之平行C++不規則結點程式庫之實作內容. 本展環境能幫助使用者撰寫平行不規則結點計算, 並藉由物件導向技術能提升程式碼重覆利用性, 降低發展成本並減少程式碼維護之複雜度.
關鍵字:不規則結點計算,平行處理,物件導向技術

Abstract

This paper describes the implementation of a platform-independent parallel C++ unstructured mesh library. This framework can facilitate parallel programming of unstructured mesh computations, and promote code reusability, reduce developing cost, and ease code maintenance with object-oriented technology.
Keywords: Unstructured mesh computations, parallel processing, object-oriented technology.

1 Introduction

The unstructured mesh is the fundamental data structure in various irregular scientific computations. Unlike a regularly structured array, an unstructured mesh can have non-uniform distribution that adapts to the dynamic nature of the problem. As a result unstructured meshes are widely used in problems that the data or the computation structure is not uniform. For example, an unstructured mesh can be used in modeling the resonance of piezoelectric crystals, the surface of aerospace vehicles, or simulating the vortices in the superconductors while they arrange themselves in a hexagonal lattice pattern to minimize the free energy when the temperature is below a critical temperature.

1.1 Object-oriented support for unstructured mesh

Although unstructured mesh computations perform different calculations according to the systems being simulated, they use the mesh virtually the same way. A mesh point updates its stored data, which represent some physic quantities at that point, by retrieving the data from its neighbors and performing calculations on them. The 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 in a systematic way. We set up a clear interface between the library and the applications so that tedious details of maintaining an unstructured mesh can be hidden from the application programmers. The application programmers only have to work on the part that they are most familiar with – the application-dependent computation kernel. All the other details should be handled by the library. Also by separating the application from the library implementation, we can choose the most efficient library implementation in a particular hardware setting, be it a Pentium PC or a large scale parallel computer.

We implemented the interface between the unstructured mesh library and the applications using object-oriented technology. We implemented various classes within the library to capture the fundamental properties of an unstructured mesh. Users of the library can inherit the generic unstructured mesh, customize them by adding application-dependent data, and redefine generic methods inherited form the basic class to suit their needs.

The framework allows fast prototyping of new unstructured mesh applications without significant performance penalty. We will show in Sec 4 that the performance inefficiency is small when compared with the saving in code development costs. We will give the timing of an exemplary flux simulation program to justify the idea of trading efficiency for fast code development.

Our object-oriented library can be easily ported to a parallel environment. The separation of library

and application lets us choose any implementation that can maximize the efficiency of a computing hardware. As the mesh size increases, the parallelization of these scientific computations becomes inevitable. The framework can make this transition from sequentialism to parallelism much more smoothly, as we will see in Section 3.

1.2 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 [3], which supports a special class PARRAY for parallel programming, the A++/P++ Array class library [9], PC++ proposed by Lee and Gannon [7, 12], 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 [10]. 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 interprocessor communication. Examples of parallel C++ projects using this paradigm include the Mentat Runtime System [6], Concurrent Aggregates (CA) [4] by Dally et al., and VDOM by [5]. 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 [8].

The rest of the paper is organized as follows. Section 2 describes the object-oriented unstructured mesh library. Section 3 discusses the potential hazards in parallelizing the library. Section 4 gives the timing results from an exemplary program for flux simulation developed within our framework. And Section 5 concludes.

2 The unstructured mesh library

We divide the framework of unstructured mesh into three layers: generic graph layer, unstructured mesh layer, and application layer, where each layer is built on top of the previous one. The generic graph layer contains basic graph operations and describes the structure of a general directed graph. The unstructured mesh layer is built on the top of the generic graph layer, with additional functionalities specific to unstructured mesh. These two layers constitute the unstructured mesh API that application programmers can use to develop unstructured mesh codes. In Section 2.3 we will demonstrate the ease of coding in our framework by an exemplary program for flux simulation.

2.1 Generic graph layer

The generic graph class serves as the foundation of our framework from which generic unstructured mesh and application-specific unstructured mesh can be derived. The generic graph is a container class that when given a user-defined data type (as a C++ template argument), will construct a graph data type where each vortex contains a data member of the specified data type. In addition, each node will have a list of pointers to its neighboring vortices, along with other house keeping data essential to data structure integrity. The generic graph class itself has a list of vertices in the graph, and other important information like the number of vertices in the graph, etc.

The generic graph class also defines basic graph operations, including the insertion and deletion of vortices and edges. These operations are essential to building a dynamic structure. In addition, we de-

fine a traversal class that will go through every vortex of the graph and perform a specific operation on it. Note that the generic graph class only provide control mechanism for traversal (i.e. in the function `traverse` in Figure 1). The actual operation performed (i.e. the function `process` in Figure 1) is up to the users of the generic graph to decide. This concept is implemented by declaring `process` as a C++ pure virtual function. We can immediately see the advantage of code reusability with object-oriented approach since after the basic graph class has been throughly tested, any of its derived class can perform graph traversal function flawlessly. This effectively reduces the development costs of writing an unstructured mesh application.

```
template <class Data>
class Graph_vortex
{
protected:
  int visit;
  long id;
  int in_degree;
  int out_degree;
  Data data;
  Link_list<Graph_edge<Graph_vortex>*> edges;
}
template<class Data>
class Graph
{
protected:
  int counter;
  int vortex_number;
  int visit;
  Link_list<Graph_vortex<Data>*> vortices;
public:
  void add_vortex(Graph_vortex<Data> *v);
  void del_vortex(Graph_vortex<Data> *v);
  void add_edge(Graph_vortex<Data> *s, (Graph_vortex<Data> *d);
  void del_edge(Graph_vortex<Data> *s, (Graph_vortex<Data> *d);
  int get_neighbor(Graph_vortex<Data> *v, Graph_vortex<Data> ***n)
}
template <class Data>
class Graph_traversal
{
  void traverse_helper(Link_list_node<Graph_vortex<Data>*> *);
protected:
  Graph<Data> *graph;
public:
  virtual int process(Graph_vortex<Data> *) = 0;
  void traverse();
};
```

Figure 1: Generic graph classes.

2.2 Unstructured mesh layer

The unstructured mesh layer serves as a mediator between the generic graph class and the applications. It encapsulates the specific details of an unstructured mesh implementation. For the current sequential implementation, the unstructured mesh layer functions as a direct channel between the application and the generic graph, and all the function calls coming from the application layer are relayed to the generic graph through stud functions. On the other hand, a parallel implementation of the unstructured mesh layer will for example, partition the entire mesh into a set of disjoint submeshes, and map each submesh into one processor. In other words, all the details related to parallel implementation will be hidden in this intermediate layer. In either case, the change in the library itself will not change the generic graph implementation or the applications built on top of the library.

```
template <class Data>
class Umesh_node: public Graph_vortex<Data>;

template <class Data>
class Umesh: public Graph<Data>
{
protected:
  Link_list<Umesh_node<Data>*> vortices;
public:
  void add_node(Umesh_node<Data> *v);
  void del_node(Umesh_node<Data> *v);
  void add_edge(Umesh_node<Data> *s, (Umesh_node<Data> *d);
  void del_edge(Umesh_node<Data> *s, (Umesh_node<Data> *d);
  int get_neighbor(Umesh_node<Data> *v, Umesh_node<Data> ***n)
};

template <class Data>
class Umesh_traversal: public Graph_traversal<Data>
{
  void traverse_helper(Link_list_node<Umesh_node<Data>*> *);
protected:
  Umesh<Data> *graph;
public:
  virtual int process(Umesh_node<Data> *) = 0;
};
```

Figure 2: Generic unstructured mesh classes.

2.3 Application layer

The library users write application by inheriting classes from the unstructured mesh layer. The application layer consists of customized classes inherited from the unstructured mesh layer, with additional application dependent data and operations. For example, we wrote a flux simulation program using Roe scheme, with customized unstructured mesh points containing all the necessary data for computational fluid dynamic in Roe scheme.

We wrote the flux simulation program as follows. First we define the data type that will be stored in each mesh point. In our flux simulation example this is defined as `Fluxroe_node` (See Figure 3). The data type `Fluxroe_node` is then given to the container classes `Umesh` and `Umesh_node` to form the actual data types for unstructured mesh and mesh points respectively. Then we put the flux computation kernel into `process` of the traversal class `Fluxroe_traversal`, which is specific to traversing a graph consists of `Fluxroe_node` data. For the flux simulation, the `process` function for one vortex goes through every edge adjacent to this vortex, gets the data of the target vortex, use Roe scheme to compute the results, and adjust the data stored in this vortex accordingly (Figure 3).

```
class Fluxroe_node
{
public:
  double data[4];
  double solution[4];
};

typedef Umesh_node<Fluxroe_node> node_t;
typedef Umesh<Fluxroe_node> mesh_t;

class Fluxroe_traversal: public Umesh_traversal<Fluxroe_node>
{
  double roeeps;
  double gm1;
  double gamma1;
public:
  Fluxroe_traversal(Graph<Fluxroe_node> *g, double r, double g1,
    double g2):
    Umesh_traversal<Fluxroe_node>(g)
      {roeeps = r; gm1 = g1; gamma1 = g2;}
  int process(Umesh_node<Fluxroe_node> *v)
  {
    Umesh_node<Fluxroe_node> **p;
    Fluxroe_node q1, q2;
    int i, n;
    n = v->get_neighbor(&p);
    for (i = 0; i < n; i++)
    {
      q1 = v->get_data();
      q2 = p[i]->get_data();
      fluxroe_compute(&q1, &q2, roeeps, gm1, gamma1); // Roe scheme
      v->set_data(q1);
      p[i]->set_data(q2);
    }
    free(p);
    return(0);
  }
};
```

Figure 3: A flux simulation example. The entire Roe scheme for is too large to list so we illustrate the control structure in process in which the function `fluxroe_compute` will be called.

3 Data partitioning and sharing

In a parallel implementation of our library the mesh structure is partitioned into submeshes and distributed over local memories of processors. In order to effect the same computation 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 submesh may require data from other processors to complete the computation of data 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. Indeed this is the case in an unstructured mesh computation where the new data value of a mesh node is a function of its neighbors. By duplicating boundary mesh nodes to the other side of partitioning lines, computations on the local submeshes on individual processors can all be performed locally without communication. In reality, data elements may be read or updated, which raises the issues of data coherence and synchronization. We describe our approach next.

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. That is, all the data elements that are essential to the computations of the local master copies will be fetched into the local submesh on the processor which owns the master copies. As far as each master copy is concerned, there is no distinction between global and local structures. Note that we do not have the notion of global pointers because all the pointers address a local memory address, be it a master copy or a duplication. 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.

Figure 4 shows the duplication mechanism for unstructured mesh. We assume that the computation of each element in the unstructured mesh requires its neighbors.

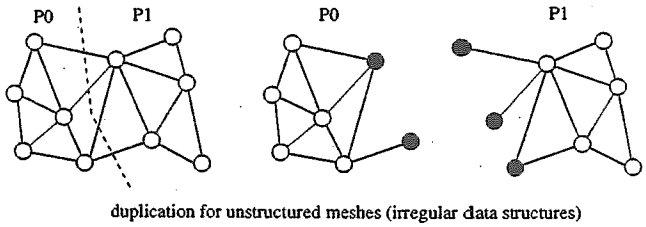

Figure 4: Duplication for distributed data structures. The duplicated data are indicated by solid black.

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. A barrier synchronization separates the duplication process from the computation, assuring that all the data are available and the computation can proceed without any further communication. This mechanism guarantees safety in a distributed environment.

The master copies must be duplicated periodically. When the data dependency and distribution are static, e.g. static unstructured meshes, we only have to allocate storage for the duplicated data once during the entire execution, then update its value once the master copy is changed. However, when the data distribution or dependency is not static, the storage for duplicated data must be dynamically allocated, or even deallocated. Nevertheless, the principle of "read-only duplication, exactly one master copy" remains the same.

4 Experimental results

4.1 Sequential experimental results

To evaluate the efficiency of our library we measure the execution time of an exemplary program developed within the framework. This application is a flux simulation program used in airfoil design [11]. We isolated the computation kernel after intensive study of the original C code, and rewrote it into a module that can be plugged into our framework. We generated the input data by using the unstructured mesh generator in [11]. This generator produces a random unstructured mesh where each edge is chosen with a fixed probability [1].

The experimental results indicate that the library only introduces a small amount of overhead. Figure 5 shows that on a Intel Pentium-133 PC the C++ version achieves up to 84% of the performance of the original C code. On a 143MHz Sun UltraSparc workstation the efficiency drops to about 62% (Figure 5). It is interesting to note that Intel Pentium CPU is more "C++ friendly" than the UltraSparc. We suspect that the function call mechanism of the Intel chip is far more efficient so that the extra function calls inevitable in object-oriented programming do not degrade the performance significantly. In any case, we are investigating the real reasons of this discrepancy and at the same time, working on ways to speed up the C++ version, for example, by using inline methods.

The preliminary timing results suggests that the object-oriented library is very beneficial for unstructured mesh application developments. The extra overhead due to object-oriented style of programming can be easily compensated by the reduced code development time. This fast prototyping capability is extremely critical when computational scientists want to have a working code as soon as possible so that they can observe the effects of a new model, and may only execute the program a few times before switching to another computation model. In such cases the ability to generate a working code quickly is far more important than the efficiency of the code generated.

4.2 Parallel experimental results

We implement a parallel version of the same airfoil code in Section 4.1. We perform experiments on a cluster consisting of 8 Pentium Pro 200MHz PC running Linux, and connected by ethernet. Figure 6 presents the timing results from our parallel implementation. We encountered considerable overhead due to the parallelization. First, we introduced complex data structures and operations for the parallelization. Therefore, even when running on a single processor (the second row of Figure 6), we can see a much larger overhead compared to the results from Section 4.1.

[1] We used 0.5 throughout the experiments.

Sun 143MHz UltraSparc						
# of mesh points	4096	9216	16384	36864	65536	147456
fluxroe.c time	0.15	0.33	0.64	1.56	2.73	6.32
fluxroe.cc time	0.25	0.54	1.06	2.45	4.33	10.17
performance ratio	0.60	0.61	0.60	0.63	0.63	0.62
Intel Pentium-133						
# of mesh points	4096	9216	16384	36864	65536	147456
fluxroe.c time	0.28	0.62	1.17	2.59	4.60	11.00
fluxroe.cc time	0.33	0.74	1.39	3.09	5.55	13.07
performance ratio	0.84	0.83	0.86	0.83	0.82	0.84

Figure 5: Timing comparison between C and C++ code for a flux simulation application. Two hardware configurations are used in the comparison: Intel Pentium-133 running Linux with 32M memory and a Sun 143MHz UltraSparc workstation with 64M memory.

Execution Time Per Traverse				
# of 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 processors	0.283	0.677	1.160	2.667
no communication	0.154	0.355	0.645	1.443
4 processors	0.144	0.335	0.623	1.395
no communication	0.082	0.171	0.312	0.713
8 processors	0.083	0.160	0.310	0.710
no communication	0.033	0.085	0.166	0.359

Figure 6: Parallel execution time per iteration, not including the initialization stage. The system has 8 Intel Pentium Pro 200 MHz.

The communication in an ethernet-based workstation cluster is expensive. From the experimental result, the program spends almost 50% of of the total time to transfer only 6% of total data among themselves (in the largest case when the number of mesh points is 147456 and the number of processor is 8). In addition, our communication protocol may not be efficient. In the current protocol, we use an empty message to indicate the end of communication between two processors. Therefore the protocol will send n^2 packages (n is the number of processors) just for the synchronization. We are working on other more efficient synchronization mechanism, especially in a loosely coupled ethernet environment. We plan to run the same code on system connected with dedicated high-speed network, and report the new finding during the conference. In addition, we used a simple orthogonal recursive bisection (ORB) to partition the mesh points. However, since the number of floating point operations needed to update a mesh point is only about 350, the efficiency of the code is very sensitive to the quality of the partitioning scheme. We will consider other good partition method can mini-

mize the number of interprocessor communications.

5 Conclusion and future improvements

The object-oriented framework makes the development of unstructured mesh applications much more easily. The users are able to write only application-dependent portion of the code without worrying about the details of maintaining the mesh. In fact after we traced and isolated the computation kernel in the flux code, it took us only hours to plug the modified kernel into the framework. This "plugin-ability" is extremely important in code reusability and reduction of code development costs.

We demonstrate that the convenience of object-orient technology does not necessarily translate into large runtime overhead. Preliminary timing results indicate that on a Intel Pentium-133 PC we measured only 16% overhead due to our OOP framework – a very small price to pay considering the reduced costs in faster code developments. Nevertheless, We are investigating all possible optimizations that can close the gap even further, including inlining and more careful object creation and releasing.

We also implemented a parallel version of our library. The data partitioning and sharing problems that might occur during the parallelization have all be properly addressed in Section 3. We will borrow the data mapping and communication classes from our previous tree library [8] and the implementation will be straightforward under object-oriented programming model. Preliminary timing results from the parallel library indicate there is still room for performance improvement. We will be working on a more efficient implementation through the use of better communication protocol and dedicated high-speed communication network.

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] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.