1-1-2007

The extensible runtime infrastructure for particle simulations with data-space management and adaptive resource allocation.

Yu Zou
University of Windsor

The Extensible Runtime Infrastructure for Particle Simulations with Data-Space Management and Adaptive Resource Allocation

by

Yu Zou

A Thesis

Submitted to the Faculty of Graduate Studies

through Computer Science

in Partial Fulfillment of the Requirements for

the Degree of Master of Science at the

University of Windsor

Windsor, Ontario, Canada

2007

Your file Votre référence
ISBN: 978-0-494-35037-9
Our file Notre référence
ISBN: 978-0-494-35037-9

Abstract

Running an adaptive particle simulation on a distributed parallel environment is a challenge, which requires dynamically periodic repartitioning during the course of computation. Repartitioning is also necessary under time and space adaptive resource allocation. This thesis presents an implementation of dynamic and adaptive resource management middleware for scientific particle simulation problems. We optimized ATOP [20] data structure to greatly reduce the total data migration time and support data cache locality. In addition, a simplified interface is designed for typical particle simulation problems, which makes the library more useable for general scientific computation. The thesis presents the implementation of an initial solution and the design of a further optimized solution for future extension.

The library had been tested on a distributed memory cluster machine with typical bench mark simulation data graphs; and experiment results show that our optimized data structure decrease mesh adaptation time 55-99% and memory management can reduce system runtime 15-25%.

Dedication

To

My wife, my daughter

and

in memory of my parents

Acknowledgement

I would like to acknowledge my gratefulness to Dr. A.C. Sodan for her supervision during this thesis work. Without her guidance and help, it would not be possible for me to finish this work. I also thank Lin Han for his work on ATOP which provides a start point for my work. I appreciate my committee members: Dr. Dan Wu, Dr. Wai Ling Yee and Dr. Xianbu Yuan, who have been kind and patient during the time when I present my thesis work.

Finally, I thank my family, my wife and my lovely daughter, for their consistent support and spiritual encouragement.

Table of Content

APPENDIX

List of Tables

List of Figures

1 Introduction

Particle simulation is used for the dynamic representation of a wide range of natural phenomena such as Computation Fluid Dynamics (CFD) and Computational Electro Magnetics (CEM) and so on. Because of the big amount of data for the class of problems, which can be millions of nodes in one mesh, simulations of these problems usually need be executed on NUMA machines [7], like clusters or on a grid in order to overcome the physical limitation of processing capability. These scientific simulations also require that periodic repartitioning happens dynamically throughout the computation course. The repartitions must be computed to minimize both the inter-processor communication cost incurred during the iterative mesh-based computation and the data redistribution costs required to balance the load. However repartitioning and re-distributing data for scaled problems can also be very expensive, and the situation becomes even worse when the simulations are executed in a time-shared and space-shared environment. So a good adaptive resource allocation runtime approach is critical to reach a high throughput and better utilization of resources.

ATOP [20] provides an effective approach to solve the problem. First, it uses over-partitioning (pre-partitioning) of data at the very beginning or when it is necessary. When unbalanced load happens during a computation process, the system can be re-balanced by directly distributing current partitions among available nodes other than repartitioning data from scratch. Second, ATOP always uses as many processes as available processors in order to be adaptive to the environment. This way leads to better

space shared and time shared resource management. However, ATOP was not yet a fully-fledged middle-ware system: It does not allow dynamically switch between space-shared and time-shared adaptation; and it also requires searching to access to each node which adds huge overhead during migration phase. All of these weaknesses will be fully overcome in our new middleware system by applying new system design and implementation.

A good load balance algorithm balances work load among processors in a distributed environment, and keeps as few edge-cuts as possible among partitions/processors in order to avoid unnecessary communication among processors[6][9]. This can be considered as data locality management mechanism at distributed memory level by trying to keep data in local memory. On the other hand, data locality in a NUMA system can also be managed at cache level. We implement our data cache locality management by putting each partition data into a continuous aligned memory block and making each block fit into the cache of local processors, which reduces cache misses during the computation process. Our test shows that using inline partition data memory block can speed computation time up to 20%.

A desirable property of a run-time system is its ability of providing services with transparency. Then users are able to use it without totally re-write their programs. The interface provided in our project hides all implementation details and provides users a set of simple functions for a specific application area, which also adds to our middleware reusability and extendibility.

The structure of this thesis is arranged the following way: In Chapter 2, we will review particle simulations and dynamic resource allocation; Chapter 3 will cover data locality management, and Chapter 4 focuses on our approach; and finally experiment and result are to be discussed in Chapter 5. Chapter 6 will show future possible optimizations and Chapter 7 covers summary and conclusion respectively.

2 Particle Simulations

A particle system usually refers to a computer technique to simulate various natural and scientific processes, which are difficult to reproduce with traditional rendering techniques. Examples of such phenomena which are commonly done with particle systems include fire, explosions, smoke, flowing water, sparks, falling leaves, clouds, fog, snow, dust, meteor tails, etc.

Typically, particles are objects that have mass, position, velocity, and respond to forces, but have no spatial extent. A typical particle data structure from users' view may look like:

```
typedef struct{
     float m;    /* mass */
     float *x;   /* position vector */
     float *v;   /* velocity vector */
     float *f;   /* force accumulator */
     int *nbors /* Adjacent particle */
} *Particle;
```

Then for a whole particle system may look like:

```
typedef struct{
     Particle *p;      /* array of pointers to particles */
     int n;            /* number of particles */
     float t;          /* simulation clock */
} *ParticleSystem;
```

Besides the particle data structure, a particle system also includes a solver, which is a simulation function applied to the particle system. The solver may have interface like:

```
void Calculate (ParticleSystem, Particle);
```

Inside this function, the solver computes new values for each particle for the next simulation clock. Position values, velocity values and force values of each particle will be

updated according to some computation functions for the specific simulation. For example, the new position values for a particle $\mathbf{P}_k$ can be expressed as a function of its own mass, current position, current velocity, forces and its neighbor's positions:

$$\mathbf{P}_k \text{ -> x [i] } = f(\mathbf{P}_k\text{->m, } \mathbf{P}_k \text{ -> x[i], } \mathbf{P}_k \text{ ->v[i], } \mathbf{P}_k \text{ -> nbor[i], } \mathbf{P}_k \text{ -> f[i] })$$

Similarly, new values of velocity for the particle can also be expressed as functions of positions, velocity, and forces, etc.

$$\mathbf{P}_k \text{ -> v [i] } = f(\mathbf{P}_k\text{->m, } \mathbf{P}_k \text{ -> x[i], } \mathbf{P}_k \text{ ->v[i], } \mathbf{P}_k \text{ -> f[i] })$$

2.1 Typical Particle Simulation Phases

For large-scale scientific particle simulations, efficient execution on parallel computers requires periodic repartitioning of the underlying computational mesh. This repartitioning should minimize both the inter-processor communications incurred in the iterative mesh-based computation and the data redistribution costs required to balance the load. Figure 1 illustrates the typical steps involved in the execution of adaptive mesh-based simulations on parallel computers. And Figure 2 shows a 2D structured mesh with 4 partitions. Initially, the data mesh is distributed among the processors according to a specific load-balancing scheme. Then a number of iterations of a simulation computation are performed in parallel. After that, either mesh adaptation happens as local processors refine or redefine their local regions of mesh that leads to unbalanced work load among processors in the whole system. Then re-partitioning based on the current mesh and system work load is computed and mesh data is redistributed among processors

respectively. The simulation can then enter another simulation iteration phase until either more unbalanced load happened or the simulation terminate.

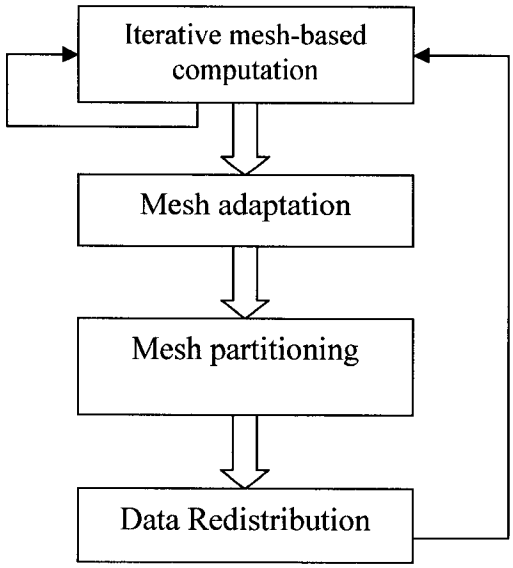

Figure 1 Typical Simulation Phases

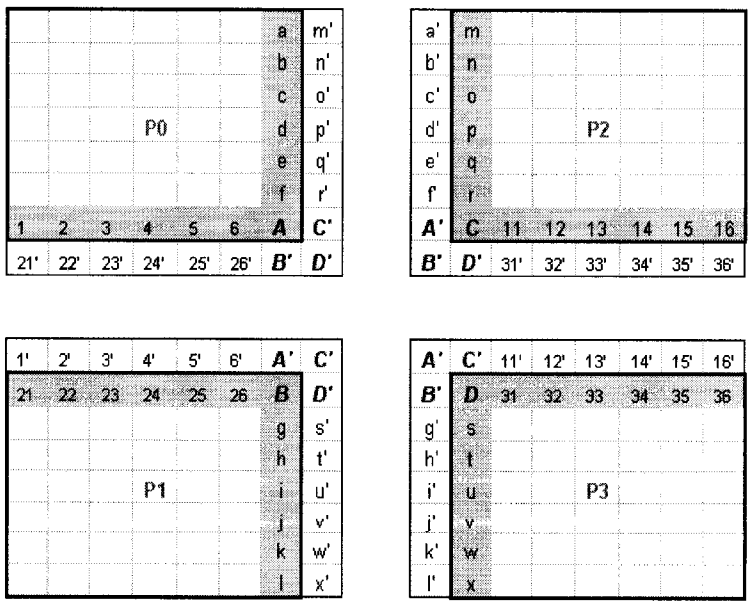

Figure 2 2D Structured Mesh with 4 Partitions

2.2 Cost Model

Considering the general processing steps illustrated in the above figure, and each round of executing includes a number of iterations of the simulation, a mesh adaptation, and load-balancing which make repartitioning and redistribution, then every round of executing will have the run time like:

$$n * (T_{cmp} + T_{cmu}) + T_{repar} + T_{dist} \quad (1)$$

$n * (T_{cmp} + T_{cmu}) + T_{repar} + T_{dist}$

Where n stands for the number of iterations for simulation computation, T_{cmp} is the run time to perform the computation for a single of the simulation, T_{cmu} is the time caused by the communication for a single iteration, and T_{repar} and T_{dist} stand for the run time of repartitioning and data redistribution respectively in a round [20].

Furthermore, communication time among processors depends on the number of edge-cut of the partitioning if we ignore the speed of physical computer. This can be expressed as a function of edge-cut:

$$T_{cmu} = f(E_{cut}) \quad (2)$$

On the other hand, data redistribution time can be described as a function of total amount of data to be moved in order to rebuild the new balanced load among processors, then we have:

$$T_{dist} = g(V_{mov}) \quad (3)$$

Combine formulas (1), (2), (3) we get total run time:

$$n * (T_{cmp} + f(E_{cut})) + T_{repar} + g(V_{mov}) \quad (4)$$

Adaptive repartitioning affects all of the terms in above formula (4). The quality of the new partitioning influences T_{cmp}. The number of edge-cuts of a new partitioning affects the inter-processor communications time. The data redistribution time is dependent on the total amount of data that is required to be moved in order to rebuild a balanced load among processors.

Generally those current adaptive repartitioning schemes [4][5][12][13][14][15][16][17][18][21][22] tend to be very fast and well balanced, which are dependent on the partitioning algorithm. Computation time not only depends on the simulation itself, it also depends on how efficient the runtime system can be. This is usually related to system architecture, memory management and other implementation details. From the perspective of partitioning and repartitioning, both $f(E_{cut})$ and $g(V_{mov})$ can seriously affect parallel run times and drive down parallel efficiencies. Therefore, it is critical for adaptive partitioning schemes to attempt to minimize both the edge-cut and the data redistribution cost when computing the new partitioning. From this point, adaptive graph partitioning is a multi-objective optimization problem.

3 Partitioning Algorithms and Adaptive Resource Allocation

3.1 Partitioning Algorithms

Since it is hard to have both $f(E_{cut})$ and $g(V_{mov})$ minimized, adaptive partitioning algorithms can be classified into two categories: The first category is to focus on minimizing the edge-cut and to minimize the data redistribution only as a secondary objective [12][18][22]. They make the best effort to compute a new partitioning from scratch and then attempt to intelligently remap the sub-domain labels to those of the original partitioning in order to reduce the data redistribution costs. Some state-of-the-art graph partitioners are used to compute the partitions, the resulting edge-cut tends to be extremely good. However, since there is no guarantee to how similar the new partitioning will be to the original partitioning, data redistribution costs can be high, even after remapping. The second category is to focus on minimizing the data redistribution cost and to minimize the edge-cut as a secondary objective [4][14][15][16][21]. These schemes attempt to perturb the original partitioning just enough so as to balance it. So they usually lead to low data redistribution costs, especially when the partitioning is only slightly imbalanced. However, it can result in higher edge-cuts than partitioning from scratch methods because perturbing a partitioning in this way also tends to adversely affect its quality.

ATOP [20] is more like an approach that combines those two categories. It first over-partitions the whole graph and then during the load re-balancing phase it can redistribute data without re-partitioning from the scratch. Meanwhile, it still provides the ability to re-partition from scratch when a higher partitioning quality is needed. From

their test result, ATOP shows a moderate increase of the number of edge cuts and it can decrease adaptation time to 20%-30%.

3.2 Work Load Distribution Algorithms

Partitioning is the first step of adaptive load balancing. After partitioning, data needs to be distributed to corresponding processors. Depending on the location of the load balancer, load balancing algorithms can be categorized as centralized or distributed. If the load balancer is located in one processor, and it has global load information of all nodes and can initiate the process of work load balancing, we call it a centralized load balancing approach. On the other hand, if each node has a load balancer which can broadcast load information to its neighbors or all other nodes, we call it a distributed load balancing approach. Because there must be one master processor to take care of the processing of load balancing, centralized approaches have limitations of scalability and are more suitable for a small number of nodes [25]. Performance of centralized approaches degrades along with the increased number of processors. Distributed approaches can be more scalable, but since global load information spreads among nodes and a node may only broadcast its load information to its neighbors, the whole system is lacking a global load profile, making these approaches less efficient as centralized approaches.

Depending on when the work load is redistributed to nodes, again load balancing algorithms can be divided into synchronous and asynchronous. Synchronous load balancing requires all nodes stopping to do other work and instead taking part in the load

balancing process. After partitioning and data redistribution each node resumes its other work. Asynchronous load balancing doesn't require that all nodes stop simultaneous for load balancing. Usually it uses work-stealing algorithms to redistribute work among nodes asynchronously, which then provides the advantage of latency hiding between computation and adaptation [3]. The disadvantage of this approach is that it also has worse load balancing quality, compared to the synchronous one.

3.3 Zoltan Partitioning and Load Balance Library

Zoltan [2] is a portable library that provides set of load balance and partition tools based on MPI. It can also run on different physical platforms. ATOP uses it as partitioner and migration tool. This makes ATOP more useable for different environments. Moreover, Zoltan supports a wide range of partitioning algorithm such as Recursive Coordinate Bisection (RCB), Recursive Inertial Bisection (RIB), and incorporates with JOSTLE and ParMETIS[10]. ATOP uses both Zoltan and ParMETIS; this fits the requirement that our system should be extensible when users need to tune it for their special usages.

Zoltan does not have special requirements on structures of nodes and meshes. However, users do have to provide global information and local information of the whole mesh and individual nodes by setting up a series of call-back functions. For example, users have to provide the total number of nodes of the whole mesh, global and local id of each node, data packing and unpacking, neighboring information of each node and so on. Somehow this is still inconvenient and time consuming for users. This is where our interfaces in Section 4.4 make improvement.

3.4 Adaptive Resource Allocation Systems

Adaptive resource allocation can get better response time and resource utilization in a time-shared and space-shared environment. Running a parallel application with adaptive resource allocation also needs cooperation from the application, which has to be malleable [22], i.e. the application needs to be adaptive to changes of available processors during the runtime. ATOP's main goal is to provide support to make simulations malleable and able to run in a shared environment with adaptive resource allocation.

Adaptation for a simulation is actually a process of load balancing, which typically needs two components: partitioner and load balancer. To make the process adaptive to a time-shared and space-shared environment, two more components are necessary: resource monitor and job scheduler. See Figure 2. So not only does adaptation happen when it is needed from the simulation phases, it also happens when the runtime environment changes. When the local workload changes like extra nodes being available or new jobs having been scheduled on the system, the resource monitor or job scheduler may signal the system to start a new round of adaptation.

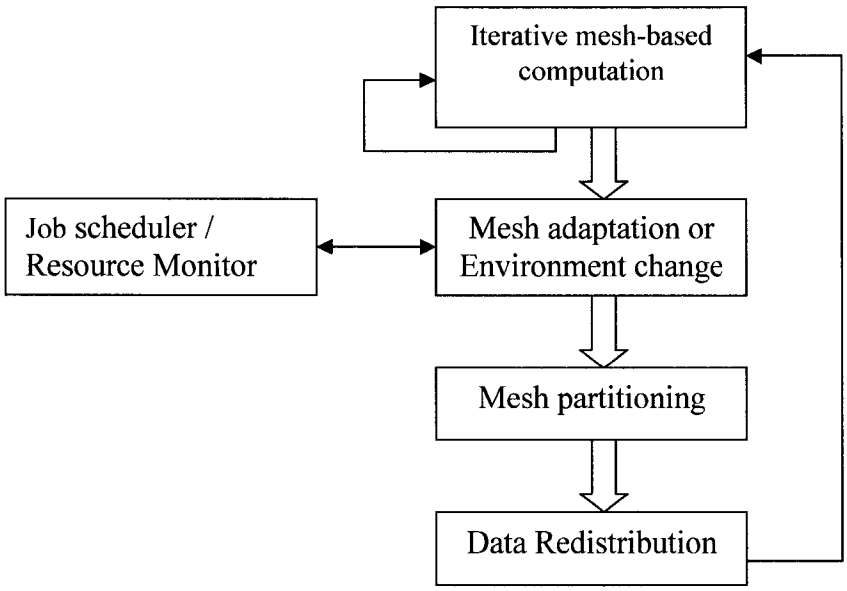

Figure 3 Simulation Phases in Shared Environment

On a NUMA system or a grid computer system, parallel execution of applications is generally based on message passing. MPI (message passing interfaces) has become the de facto parallel programming paradigm. However, so far only few runtime resource allocation systems are built on message passing. TMPI [19] and Cilk [1] tried to use threads to replace local MPI processes which are more suitable for running on a SMP shared memory machine. AMPI [8] tried to use threads to over-partition into MPI "process" and balance these threads to nodes according to load information. The major problem with these thread-based systems is that a large number of threads per node tend to result in a big thread context switch overhead, and slow down applications if communication is frequent. However, scientific particle simulations are communication intensive. After every round of mesh computation, communication for exchanging local mesh information with remote meshes is necessary.

4 Data Locality Management

In the last decades, the speed of processor doubled for every 2-3 years but memory access speed only improved 10% every year. The gap between CPU speed and memory access speed has become a bottleneck for enhancing computing power. In order to reduce this gap, modern computers resort to multilevel memory hierarchies. Usually modern computers have two levels of cache memories, and further NUMA machines have both local memory and remote memory, where local memory can be considered as the cache of the remote memory. However, computing performance improvement with this memory structure depends on whether the caches can provide data that an application needs at runtime, because cache misses will result in spending extra time to access data from higher-level memory. So application performance becomes very sensitive to the cache hits.

For particle simulation applications, improving data locality of applications will lead to better performance. From formula (4), data locality can affect all the factors of a particle simulation, especially T_{cmp} and T_{cmu} because a typical particle simulation tends to repeat hundreds and even thousands of times to compute mesh between two sequential mesh adaptations, and total number of repeated computation for a whole simulation may reach to millions. Bad data locality always slows down a simulation during running time. Even a small amount of saving time during the computation phase can contribute a big jump for the whole application.

Data locality in a NUMA machine usually can be achieved at two levels. The higher level or network level data locality can be provided between the local memory and remote memory. Better partitioning strategies keep more needed data in local nodes, and network latency and communication cost can be reduced. So, as we discussed in Chapter 2, those partitioning algorithms focus on minimizing edge-cuts among partitions [12][22] will have better data locality. In addition, this level of locality can be obtained by keeping some extra data on local nodes: besides storing those vertices that belong to the local mesh, all neighbor vertices will have a copy stored in local memory, which are called ghost cells. For example, Figure 3 shows that four processors with local meshes. Shaded cells are border nodes and labeled cells are ghost cells.

Cache locality can be provided at local processors only. On a time-shared machine, there are two factors influence cache locality: 1), several processes or threads share the same cache. One running process or thread displaces the cache content which was built by the previous running process or thread. When the previous process or thread runs again it will encounter a series of cache misses. 2), either memory fragmentations or random memory accesses will affect utilization of cache lines. In time shared environments we can not avoid processes or thread switches, but we still can obtain better cache locality by optimizing application data structure. To achieve this goal, the order of access to data is crucial. By sorting or aligning all local data in memory, all memory access patterns for this problem can become sequential as opposed to random. This allows maximum use of hardware-based performance features such as caching and automatic pre-fetching. Anyway, the topic of particle sorting is not unique but generally

ignored for current adaptive runtime systems due to the implementation complexity and performance overhead.

5 Our Approach

Our objective is to implement an efficient dynamic resource allocation middleware for scientific particle simulations. We reach this goal by finishing the following jobs in our project:

- Extend ATOP partitioning and load balancing algorithm by using optimized data structures to do direct accesses.
- Add data locality management to ATOP by aligning local data into blocks of continuous memory to provide cache locality.
- Add communication management by using border-ghost mapping to automatically communication and make communication more efficient by aligning packing and unpacking.
- Implement switching between time-share adaptation and space-shared adaptation.
- Interface makes the middleware easy to use.

5.1 Basic Concept

Definition: *Space Adaptation*: The number of processors allocated to an application can be changeable dynamically during runtime [20].

Definition: *Time Adaptation:* The time share allocated to an application can be different on different processors and can be changed dynamically during the application's execution process [20].

5.2 Architecture of the Runtime Library System

In our architecture, see Figure 4, the operating system is at the bottom and above that the MPI library is located. The ATOP load balancing algorithm and our data locality management (DLM) module lie above Zoltan and MPI, which manages memory alignment, optimization, border cells and ghost cells. We use the ATOP algorithm to over-partition the data mesh and handle dynamic adaptation when an adaptation is necessary. We redesigned internal data structures of ATOP to avoid searching problems and also make dynamic switching between time-shared and space-shared allocation possible. Beneath ATOP and DLM is the Zoltan load balancing library which provides multiple partitioning and data management tools.

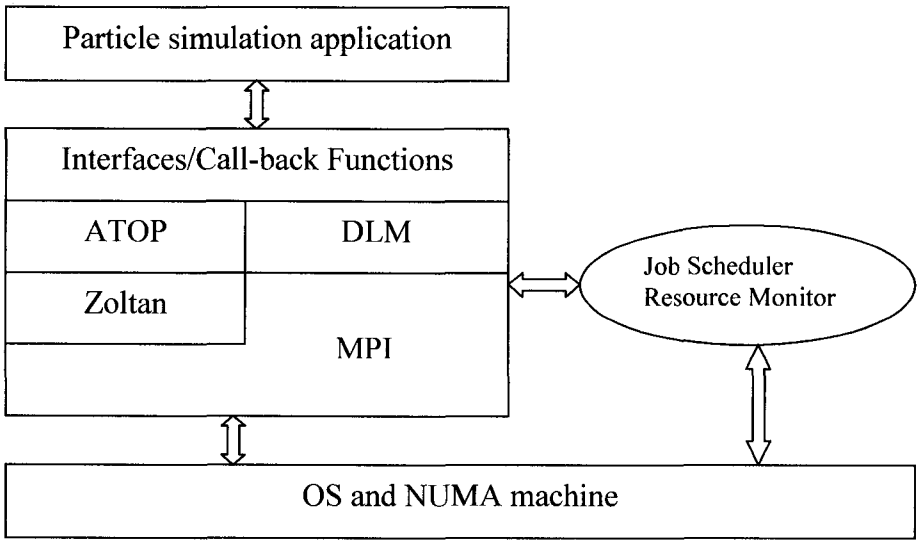

Figure 4 System Architecture

The Interfaces and call-back functions layer is above ATOP and DLM, and encapsulate all the implementation detail from the users and provide users simple and easy-to-use library functions. A good design should release application programmers

from too much attention on dynamic variations in computational load and communication patterns. So we provide a simplified set of interfaces in order to keep users from dealing those tedious lower level works such as load balancing, data management, migration and etc. Detailed interfaces will be discussed in Section 4 of this chapter.

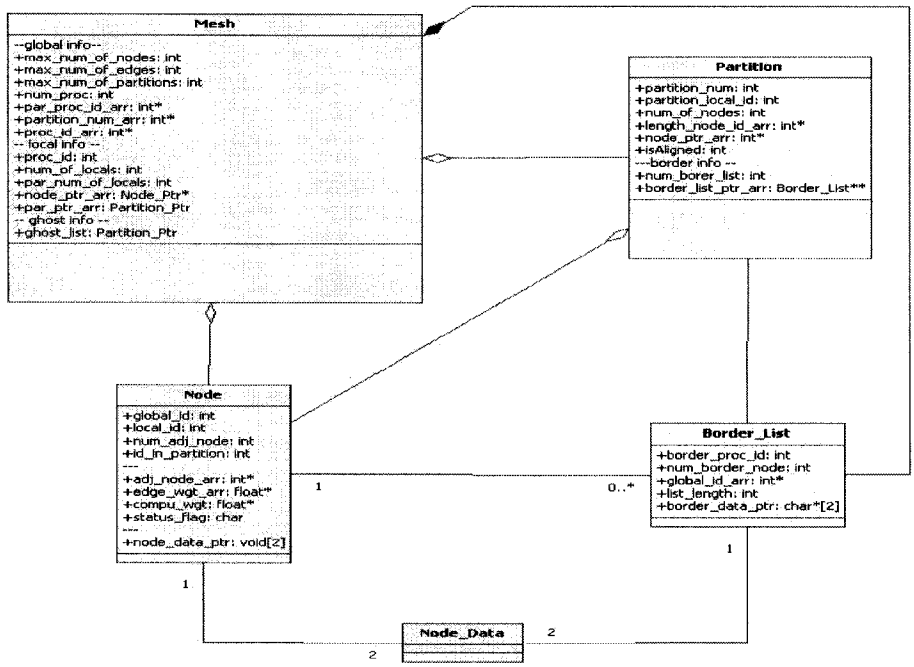

Figure 5 System Class Diagram

We use an objective-oriented technique to design and implement our system, though we still use C as programming language for this project. The whole system is illustrated as class diagram like Figure 5. From the diagram, we can see that the main data structures are ***node***, ***mesh*** and ***partition***. ***node*** contains all necessary information and related operations of a node such as global id, local id, number of adjacent nodes and etc. ***mesh*** includes all the global and local information for a mesh. Technically a ***mesh*** is an aggregation of ***partitions, nodes*** and ***border_list***. ***partition*** is an aggregation of nodes.

All related operations / functions are defined in header files *mesh.h, partition.h* and *node.h.* The public accessibilities for all data members and methods in the UML diagram means they are accessible for C programming language as long as a right header file is included.

5.3 Internal Data Structures

Internal data structures are the backbone in the system. In addition to the class diagram in Figure 6 which shows the relation among different types, more detail is needed for explanation. Some sample code is attached in APPENDIX B.

The biggest drawback in the original ATOP implementation is that it keeps nodes in a local array of nodes. During the adaptation process, outgoing nodes are removed from the node array directly; and incoming nodes are inserted into the array by looking up an empty spot in the array sequentially. This approach led to two big problems: First, the outgoing nodes leave some holes in the local node array and inserting incoming nodes into the array make the array out of order. When the system needs find a node, it has to search the node from the array one by one. Second, using an array of nodes actually assumed that each node has the same size, which makes it impossible to extend the approach when we need to include border-ghost management and communication management in our runtime system. More flexible data structures have to be designed.

In our new design, each node, partition and mesh is an independent object. Local meshes and partitions can access to nodes by direct mapping using an array of pointers.

More importantly, the index of the pointer array is exactly the global id of node. So sequentially searching a node is no long necessary because we can get access to a node directly if we know the global id of the node by obtaining a pointer to the node from the node pointer array. Moreover, this makes the system architecturally neater and enable to perform border-ghost-cell management and communication management.

5.3.1 Mesh Data Structure

Mesh data structure contains information about the number of processors, global location of each partition and node, reference to local partitions, local nodes, ghost nodes and border lists. The information can be classified into global information and local information. Global information stands for the whole data graph. Local information stands for the sub-graph located on the local processor.

1. The global information is mainly for load balancing. It shows how the current mesh is distributed among processors, including location, global id of partitions and nodes.
   - Global partition information is about where each partition locates. Suppose the graph has ***m*** partitions, integer array ***par_proc_id_arr*** of each mesh records the processor number for each partition globally. In other words, each processor will keep a copy of the table and update it during each data redistribution time.

partition_global_id	0	1	2	3	.....	m-3	m-2	m-1
par_proc_id_arr	0	1	1	1		15	15	15

   - Similarly, global node information is about where each node locates, which tells us on which processor a node is located.

node global id	0	1	2	3	.....	n-3	n-2	n-1
proc_id_arr	2	0	1	0		2	0	1
partition_num_arr	5	0	3	0		5	0	3

2. Local information describes how partitions and nodes are stored in the local processor, which is very important for computation and communication. By having an array of pointers to each local partitions and nodes with their global id as index, only constant time (***O***(0)) is needed to find a partition or a node. Furthermore, mapping arrays between local id and global id makes it possible to handle both partitions and nodes without losing global information.

- The local information for partitions in a ***mesh*** is illustrated by Figure 6. The ***par_local_global_map*** is actually an array which contains all local partitions' global id. The indexes of the array stand for local ids of partitions. This means that we can get each partition's global id as long as we know a partition's local id.

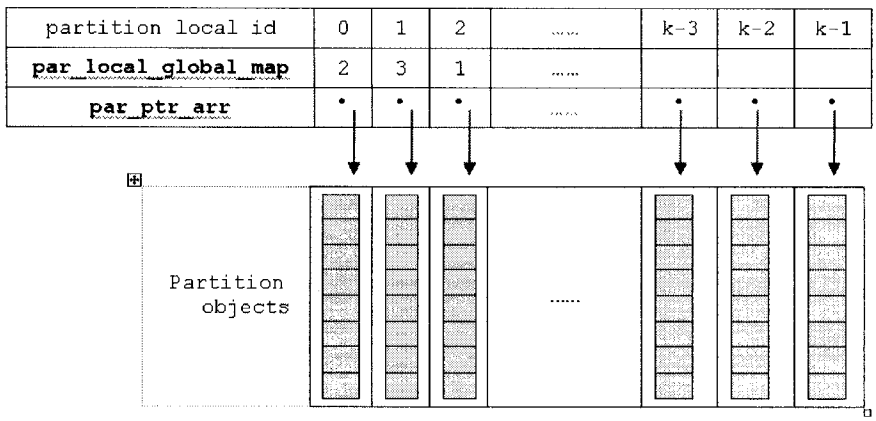

Figure 6 Local information 1

- Local information for nodes in a mesh is illustrated by Figure 7. The data member ***local_global_map*** is an array which contains the global id of each local node. The indexes of the array stand for the local ids of local nodes.

Local id	0	1	2	.....	m-3	m-2	m-1
local_global_map	3	24	2	.....	x	y	z

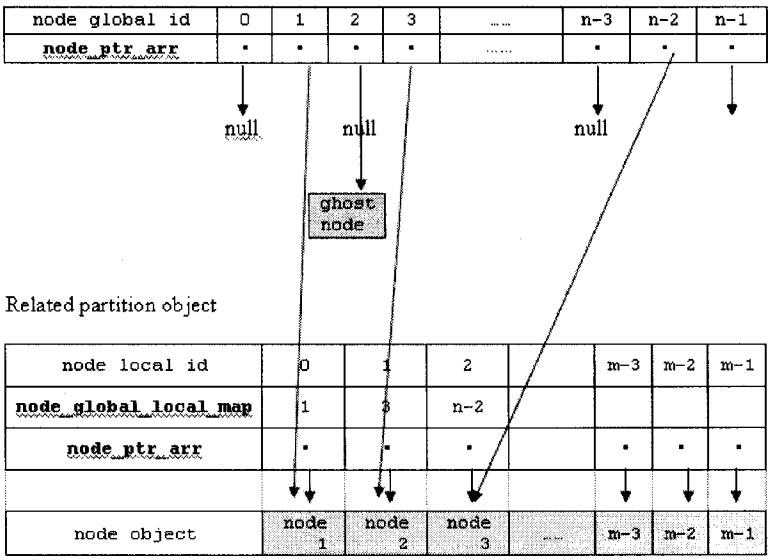

Figure 7 Local information 2

5.3.2 Partition Structure

Partition is the basic load balancing unit in over-partitioning algorithm, which stands for a sub-graph of the local mesh and the local mesh is a sub-graph of the whole data graph.

Partitions track all information about nodes which belong to it, see Figure 7. The array ***node_global_id_arr*** records all local nodes' global id. Meanwhile the pointer array ***node_ptr_arr*** contains all pointers that point to all the nodes of the partition. The array is used during the computation stage when we need traverse nodes one by one for each partition.

5.3.3 Node Structure

Node data structure stands for a particle in a particle system or a vertex in a data graph. An internal node data structure includes basic information about itself: global id, local id, neighbor node information and edge-weight information. All of these are necessary for Zoltan to manage partitioning and migration. Considering migration cost and system scalability, the node structure should be kept as small as possible.

The node structure includes two parts: the first is the general information for partition and migration; the second part is the node data for the computation. A node may be a normal node, border node, or ghost node. A normal node is illustrated in Figure 8. Border nodes and ghost nodes need more effort in handling because borders and ghosts need to exchange information among processors. In order to avoid unnecessary data copies for the communication phase, border nodes' data is stored directly in the border list (see Figure 9), except borders which are adjacent to more than one remote processor (see Figure 10). Data member ***status_flag*** is used to track the type of current node:

1. ***status_flag*** data member of a node is of char type with a byte long, which is used to record a node's status. Now only bit 0 to 3 are used:
   - Bit 0, stands for if the node is aligned. If this bit is set to 1, it means that current node is in a continuous memory block.
   - Bit 1, stands for if the node is a ghost node. If the bit is set to 1, it means the current node is a ghost node.

- Bit 2, stands for if the node is a border node. If the bit is set to 1, it means the current node is a border node.
- Bit 3, used by border nodes. If both this bit and bit 2 are set to 1, it means that the current node is bordered to multiple processors.

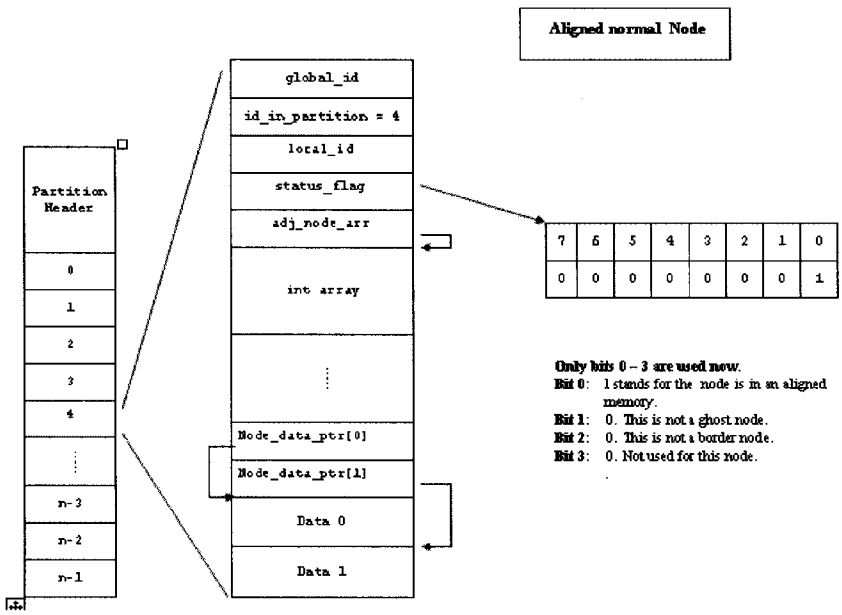

Figure 8 Aligned Normal Nodes

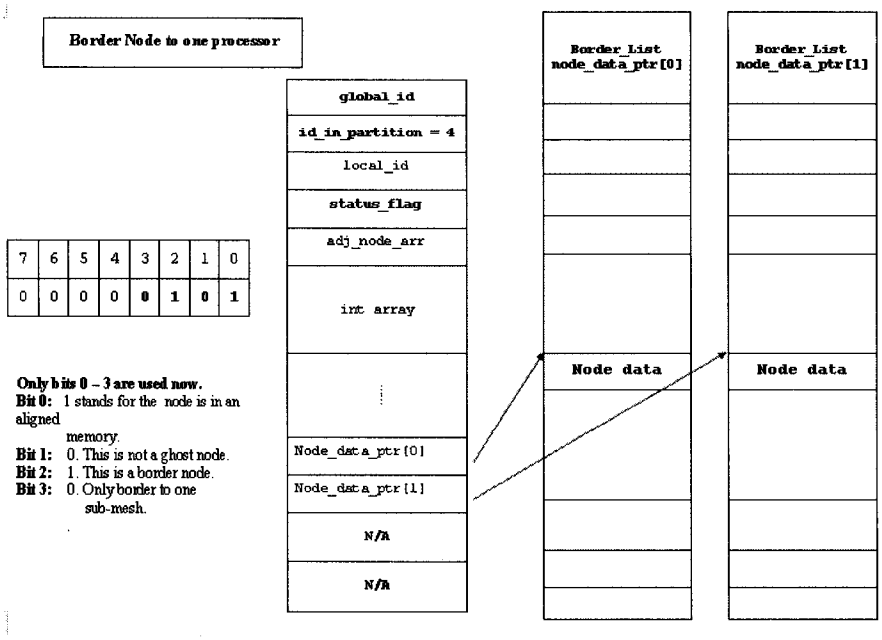

Figure 9 Aligned Nodes Bordered to One Processor

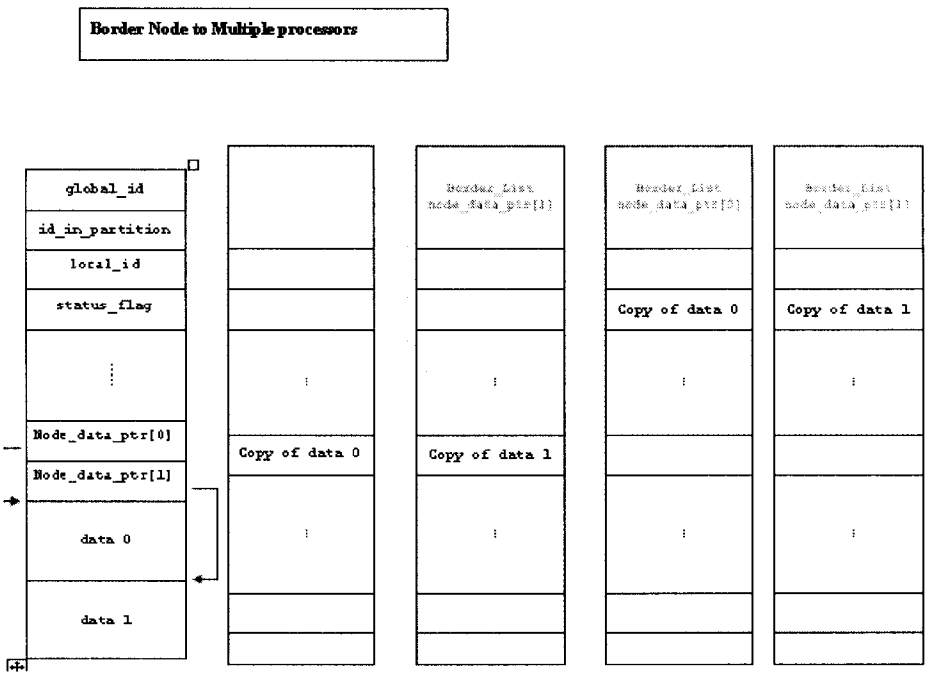

Figure 10 Aligned Nodes Bordered to Multiple Processors

5.3.4 Borders and Ghosts

Border and Ghost cell management are very important to reduce network latency during computation process. Borders are those nodes which have neighbor nodes located on remote processors, and ghosts are copies of border nodes on other processors. In other words, they are simply images of border nodes on other processors. Border and ghost cell management is used for communication because after each computation round, we need to update values of ghost cells for the next computation round. Updated values of border cell will be sent to corresponding remote processors to update those ghost cells. Borders and ghost cells need to be rebuilt after each mesh adaptation.

Border and ghost cell management includes two stages: build/rebuild stage and update stage. After each mesh adaptation, global location of some nodes will be changed, some border nodes will no longer be border nodes as they do not have any neighbors on remote processors; some local nodes become new border nodes because their neighbor nodes have been migrated to other processors. So a rebuild needs to be done before starting the next round of simulation computation. During simulation, after each round of computation, the data values for each node are changed but these changes only take place for local nodes. Since border nodes change too, their images in ghost cells need to be update before next round of computation.

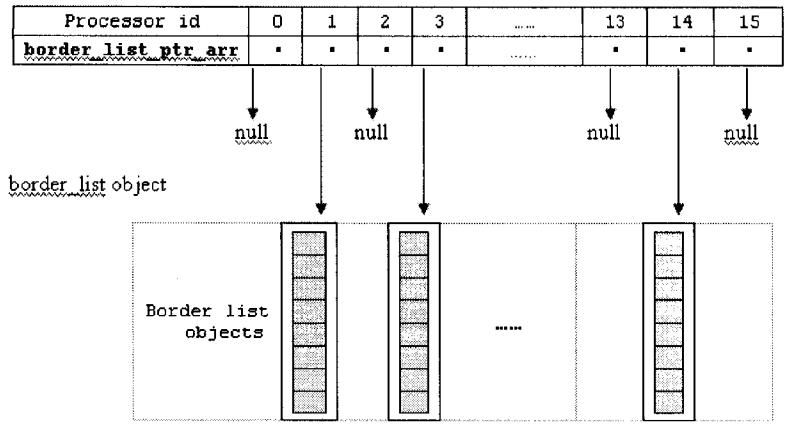

Figure 11 Border List Structure

To maximize performance of our system each local mesh keeps a border table, see Figure 11, which is a number of arrays with each array standing for border information to a specific remote processor. On the other hand, each local mesh has only one array of ghost nodes. Technically the array of ghost nodes has the same structure as other normal partitions except ghost nodes inside the partition need special treatment for their data, see Figure12.

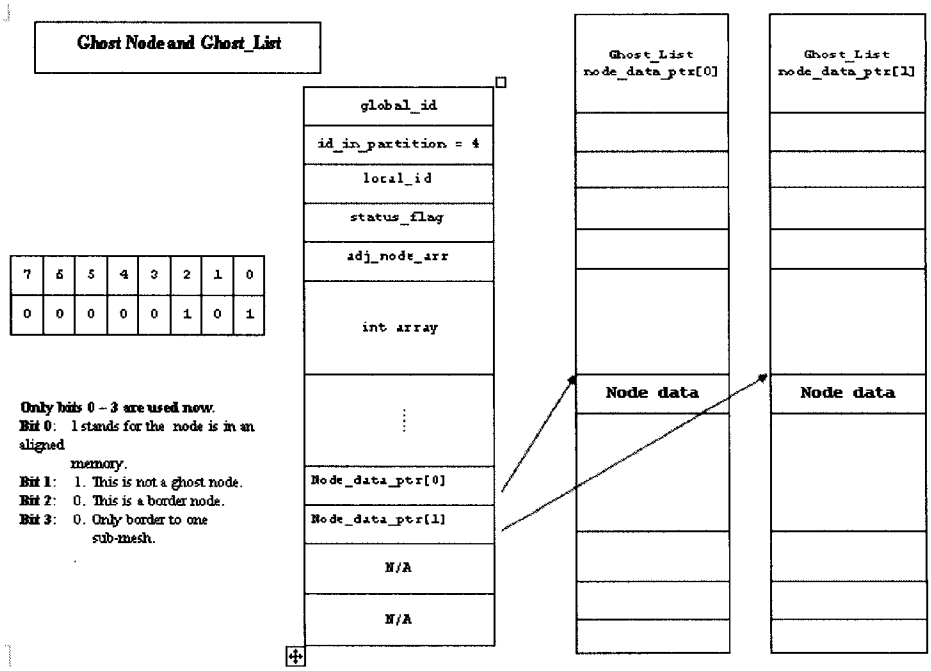

Figure 12 Ghost Nodes

5.4 Interfaces

To separate general user-level application data structures from our internal data structures, we design and implement a set of simplified interfaces, seeing Figure 13. By hiding all the complicated details like environment initialization, data migration and load balancing, simplified interfaces are provided. This approach can also make the runtime system more extendable and maintainable.

These interfaces have been loosely tied to users' development style. Users can have their own node data structures without worrying about how the system handles them. For example, particle node data can be as simple as just an integer or a user-defined structure which includes multiple arrays of data to describe a node's position, force and

relation to other nodes. The requirement from the system is that the user has to bind the user-defined information to the system by calling a couple of call-back functions. One advantage of our interfaces is that users do not need deal with communication manually. All communication among processors are handled automatically by the library.

Applying the interfaces to real simulations is simple. A programming example of particle simulation is shown in Appendix 3, which is also the basic application used for our experiments in Chapter 6. Using the library generally includes the following steps:

1) First, a user defines his or her node data structure and ties it to the system by using the callback function `ATOP_Set_Data_Size_Fn(&node_data_size);`
2) Second, the user defines his or her computation function and ties it to the system by using callback function

```
ATOP_SET_MESH_COMPUTE_FN(ATOP_MESH_COMPUTE_FN my_calculator);
```

3) Then the mesh, and computation environment is initialized;
4) An adaptation can be performed by calling `ATOP_ADAPTATION(Mesh_Ptr mesh, Int* comput_weight_arr);`
5) Repeatedly `ATOP_MESH_COMPUTE(ATOP_MESH_PTR)` is called to do mesh computation; or the user goes back to step 4) to make another adaptation if it is necessary; or goes to step 6) if the simulation is finished;
6) The simulation is finished by releasing environment resources.

```
struct ATOP_NODE{
       int    global_id;
       int    num_adj_node;
       int*   adj_node_arr;
       int    proc_id;
       int    isGhost;
       void*  node_data_ptr[2];
};

typedef struct ATOP_NODE ATOP_NODE;
typedef struct ATOP_NODE *ATOP_NODE_PTR;


struct ATOP_MESH
{
       int    num_of_global_nodes;
       int    local_num_of_nodes;
       int    num_proc;
       int    proc_id;
       int    new_data_index;
       int    old_data_index;
};

typedef struct ATOP_MESH ATOP_MESH;
typedef struct ATOP_MESH* ATOP_MESH_PTR;

typedef void (*ATOP_MESH_COMPUTE_FN)(ATOP_MESH_PTR, ATOP_NODE_PTR);

/*********** END USER INTERFACE FUNCTIONS *********************/
int    ATOP_ENVIRON_INIT(int argc, char **argv);
void   ATOP_MESH_DESTROY(ATOP_MESH_PTR my_mesh_ptr);
void   ATOP_ENVIRON_DESTROY();
void   ATOP_MESH_DISPLAY();

ATOP_MESH_PTR  ATOP_CREATE_MESH(char* graph_name, int num_partitions);

void   ATOP_SET_DATA_SIZE_FN(int(*data_size_function_name)());
void   ATOP_SET_MESH_COMPUTE_FN(ATOP_MESH_COMPUTE_FN my_calculator);
void   ATOP_ADAPTATION (ATOP_MESH mesh, int* weight_arr)
void   ATOP_MESH_COMPUTE(ATOP_MESH_PTR);

ATOP_NODE_PTR       ATOP_GET_NODE(ATOP_MESH_PTR mesh_ptr, int global_id);
```

Figure 13 Interfaces

5.5 Memory Alignment

Memory alignment is the core of the data locality management. For a mesh or a graph, the nodes usually are generated dynamically, which means those nodes are always spread in memory, see Figure 14. But after finishing memory alignment, all nodes for a partition are located in one memory block and can be accessed sequentially. During the partitioning stage, the number of partitions for the whole global mesh can be chosen, and the memory size of a partition can be adjusted. The appropriate size for the partition located in a continuous memory block can take advantage of the cache performance and finally enhance throughput of the whole application.

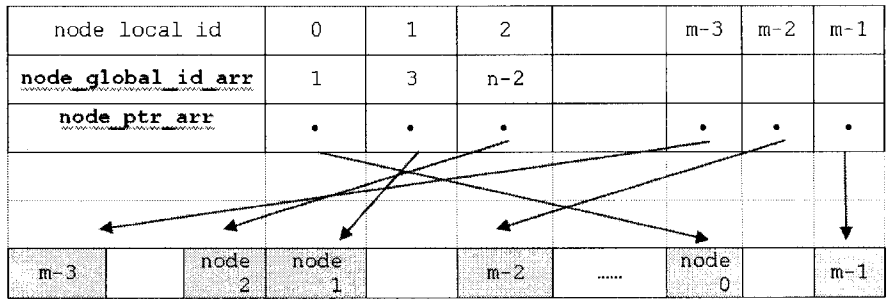

Figure 14 Un-aligned Partitions

When aligning memory for a partition, the memory management module will only put the partition header information and all corresponding nodes in one continuous memory block. A good reason for this structure is future extension. Right now Zoltan can only migrate nodes but can not migrate partitions directly. However, keeping all partition information together provides an opportunity to migrate partitions individually in the future. After alignment a partition will look like Figure 15.

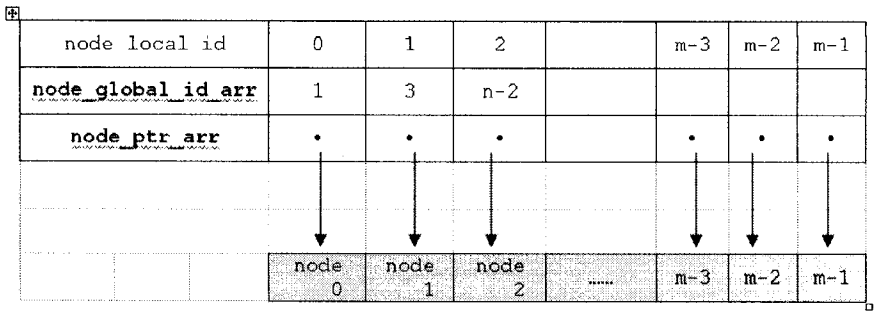

Figure 15 Aligned partitions

6 Experimental Results

6.1 Experimental Environment

We perform all our experiments on an IBM cluster, which includes 16 nodes, and each node contains dual Intel Xeon processors with 512 Mbytes of RAM. All nodes have 2.2 GHz CPU except the two nodes which have 2.4 GHz CPUs. Nodes are connected by Myrinet high-speed interconnect. The operating system is Debian Linux with 2.6 Kernel on each node, the MPI package we used is MPICH 1.27, the Zoltan version is 2.1 and the ParMETIS is version 3.1, and K-way partition algorithm is used.

Graphs	**|V|**	**|E|**	**Description**
3elt	4720	13722	2D finite element mesh
4elt	15606	45878	2D finite element mesh
wing	62032	121544	3 finite element mesh
brack2	62631	366559	3 finite element mesh
finan512	74752	261120	N/A
wave	156317	1059331	N/A

Table 1 Benchmark Graphs

We chose the benchmark graphs from University of Greenwich Graph Partition Archive [26], all with Chaco graph file format. This graph archive contains various types of graphs which fit our test plan. For better comparison we choose those test graphs which were used in [20]. Another bigger graph ***wave***, which has twice as many vertices

as graph finan512 and three times as many edges as graph brack2, was chosen to test the scalability of our runtime system.

6.2 Adaptation Performance

As we mentioned, ATOP is used in our system to manage partitioning and load balancing, however, new data structures were designed to achieve higher performance and avoid un-necessary search for nodes which existed in the original ATOP implementation. So we use the same set of graphs used in [20] to do our experiments in a space-shared environment and a time-shared environment to get comparable experimental results.

6.2.1 Adaptation Experiments in a Space-shared Environment

Both Table 2 and Table 3 show experiments in a space-shared environment. When using ATOP over-partitioning algorithm, graphs were partitioned into 8 partitions initially and number of processors changed in an order 4 – 8 – 4 – 2. Shaded rows are data from [20], and all data is in seconds. The init time includes time for creating graphs by reading graph files from hard drives and time for initially partitioning and migration to 4 processors. The adaptation time is the total adaptation time for the three adaptations (4->8, 8->4 and 4->2).

The data shows big jumps in performance. Table 2 shows that the init time for over-partitioning can be shortened by 93% to 98.7% from these graphs, and adaptation time for over-partitioning can be improved by 90% to 96%. On the other hand, Table 3

illustrates that init time for partitioning from scratch can speed up by 94% to 99%. For over-partitioning, the adaptation time for partitioning from scratch can be improved by 63% to 75%. The difference of edge cuts is because we used a new version of Zoltan.

Graphs	Overall Edge Cuts				Init Time	Adaptation Time
	4	8	4	2		
wing	2390	3261	2390	1858	202.3	25.8
	2428	3287	2428	1858	12.1	2.3
	38	26	38	0	-189.9	-23.5
	Speedup				**94%**	**91.1%**
brack2	5047	8528	5047	4069	204.1	26
	3893	8197	3893	2907	12.7	2.4
	-1154	-331	-1154	-1162	-191.4	-23.6
	Speedup				**93%**	**90.7%**
finan512	405	648	405	324	296.0	49
	405	648	405	324	3.9	2.0
	0	0	0	0	-292.1	-47
	Speedup				**98.7%**	**95.9%**

Table 2 Adaptation by over partitioning in space-shared environment with 8 partitions

Table 4 and Table 5 illustrate init time and adaptation time comparison for 16 partitions and experiments executed on 16 processors. More graphs are tested and the adaptations follow 8 –16 – 4 –2 processor sequence. Init time includes initial mesh generation, partitioning into 16 partitions and distributing partitions to 8 processors.

Similarly the shaded data come from [20] and all the rest is from current experiments. The results show that, for small sized graphs like 3elt and 4elt, our approach does not gain any advantage. But if the size of the graphs becomes bigger, the optimized data structures contribute greatly to the whole init and adaptation performance. Figure 16 and Figure 17 illustrate our performance enhancement in graphic form, where OP stands for current results.

Graphs	**Overall Edge Cuts**				**Init Time**	**Adaptation Time**
	4	8	4	2		
wing	2129	3081	2036	947	215.3	102.7
	2325	3087	2186	999	11.8	32.3
	196	6	150	52	-203.5	-70.4
	Speedup				**94.5%**	**67.5%**
brack2	3163	8221	3113	747	223.0	88.8
	3159	8222	2972	754	12.7	33.0
	-4	1	-141	7	-210.3	-55.8
	Speedup				**94.3%**	**62.8%**
finan512	324	648	324	162	328.7	199.8
	324	648	324	162	3.9	50.8
	0	0	0	0	-324.8	-149
	Speedup				**98.8%**	**74.5%**

Table 3 Adaptation by partitioning from scratch in space-shared environment with 8 partitions

Table 6 and Figure 18 show init time and adaptation time comparison with 128 partitions by using over-partitioning. Initially the experiments were running on 8 processors, and then adapted to 16, 4 and 2 processors respectively. The results show our optimized implementation has speedup up to 99% for init time and speedup up to 96% for adaptation time. This proves that our implementation can also achieve better performance for large number of nodes. Larger number of partitions does not affect our system's performance.

	3elt		4elt		wing		brack2		finan512	
INIT TIME	0.2	**0.5**	3.3	**1.2**	210	**12.1**	222	**12.7**	312	**3.9**
AP TIME	0.1	**0.2**	0.5	**0.4**	20.3	**2.0**	22.6	**2.1**	38.0	**2.1**

Table 4 Init time and adaptation time by over-partitioning for 16 partitions

	3elt		4elt		wing		brack2		finan512	
INIT TIME	0.3	**0.3**	3.6	**1.2**	234	**11.8**	245	**12.7**	354	**3.9**
AP TIME	0.2	**0.4**	2.2	**2.2**	61.5	**34.2**	56.5	**31.1**	113	**50.5**

Table 5 Init time and adaptation time by partitioning from scratch for 16 partitions

	3elt		4elt		wing		brack2		finan512	
INIT TIME	0.4	**0.4**	4.1	**1.0**	215	**10.5**	224	**11.5**	326	**2.8**
AP TIME	0.1	**0.2**	0.4	**0.4**	17.5	**1.2**	18.1	**1.2**	30	**1.0**

Table 6 Init time and adaptation time by over-partitioning with 128 partitions

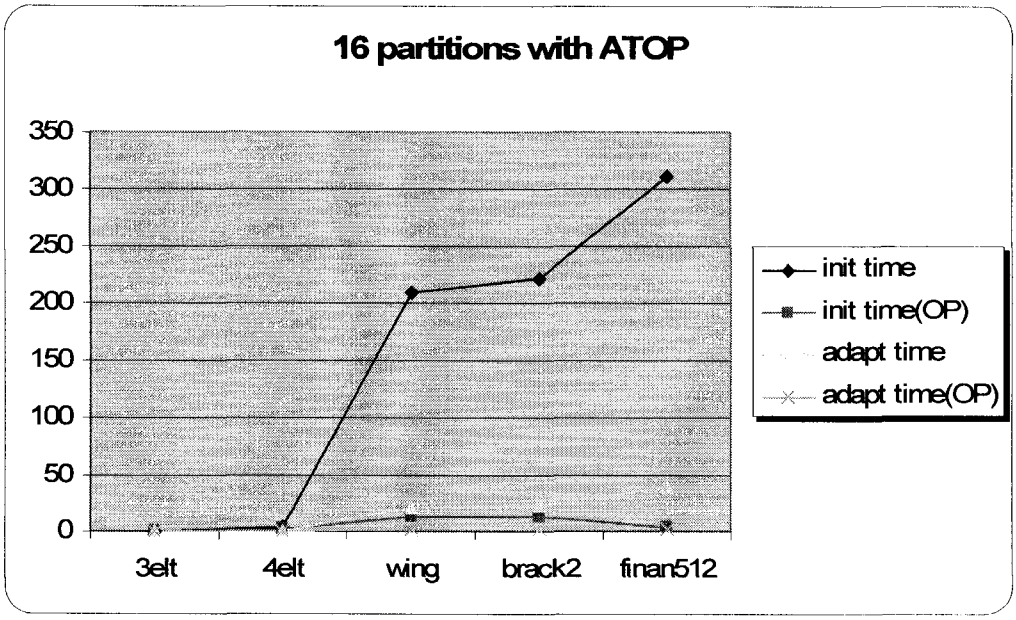

Figure 16 Comparison for 16 partitions by ATOP

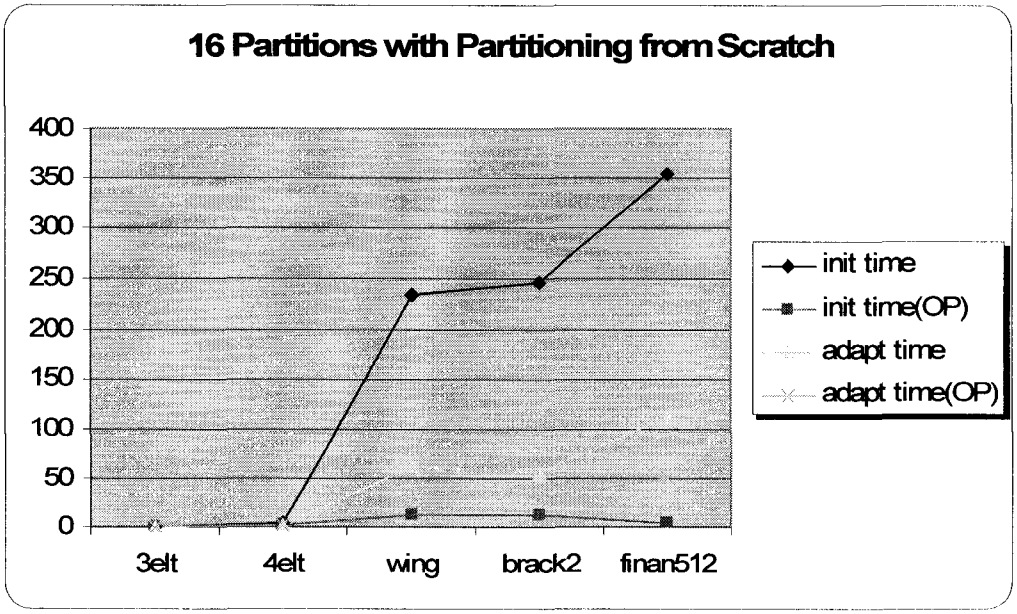

Figure 17 Comparison for 16 partitions by partitioning from scratch

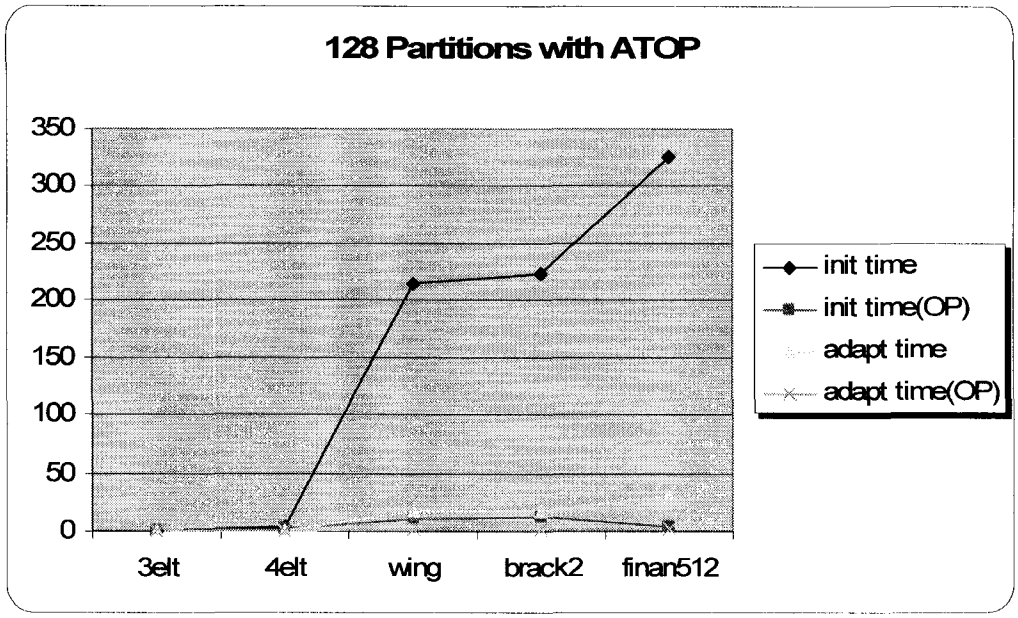

Figure 18 Comparison for 128 partitions by ATOP

6.2.2 Adaptation Experiments in a Time-shared Environment

In order to test our system in a time-shared environment, we assign each processor with different computation weight to simulate changed time shares. Initially we set up 8 processors with relative computation weights 2:2:2:2:1:1:1:1, then the weights changed to 1:1:1:1:1:1:1:1 -> 1:1:1:1:2:2:2:2 -> 1:1:1:1:2:2:3:3 respectively. Table 7 and Table 8 show the experiment results for using ATOP over-partitioning algorithm and using partitioning from scratch by using K-way partitioning respectively.

Table 7 and Table 8 show that our optimized implementation also provides improvement in a time shared environment. The bolded columns stand for the speedups. Init time gains up to 99% speedup for both over-partitioning and partitioning from scratch, as for space-shared cases. Adaptation times also show large improvements.

Adaptation time by over-partitioning in our implementation can reach a speedup to 97% and the adaptation time by using K-way partitioning from scratch can get up to 72% improvement.

	Init			1:1:1:1:1:1:1:1			1:1:1:1:2:2:2:2			1:1:1:1:2:2:3:3		
wing	221.9	10.7	**95%**	3.92	0.26	**93%**	4.58	0.24	**95%**	0.91	0.13	**85%**
brack2	222.3	12.3	**94%**	3.72	0.38	**90%**	4.48	0.72	**84%**	0.93	0.42	**55%**
finan512	347.7	2.94	**99%**	4.33	0.19	**95%**	6.02	0.21	**97%**	0.10	0.10	**0**

Table 7 Init time and adaptation time by ATOP environment with 128 partitions

	Init			1:1:1:1:1:1:1:1			1:1:1:1:2:2:2:2			1:1:1:1:2:2:3:3		
wing	226.93	10.63	**95%**	22.05	6.87	**67%**	29.49	8.22	**72%**	1.87	9.05	**-384%**
brack2	229.98	11.58	**95%**	25.80	7.02	**73%**	24.56	8.41	**66%**	2.10	10.40	**-395%**
finan512	354.52	2.83	**99%**	23.18	8.66	**63%**	24.54	10.5	**57%**	3.2	12.36	**-286%**

Table 8 Init time and adaptation time by K-way Partitioning from scratch with 128 partitions

6.3 Data Locality Management Module Performance Experiments

The locality management module plays the most important role in our system and deserves separate experiments to test it. Currently this module is implemented by using memory alignment. In order to test performance of this memory alignment, we need a test scenario which includes typical particle simulation data structures and a particle computation function.

As we discussed in Section 5.4, our implementation allows users to define their own node data structure and solver computation functions. Then they can use call-back functions to tie their own functions with our runtime system. For this experiment, we use a very simplified node data structure and computation functions like below:

```
double node_data[2];
```

$$\texttt{node\_data[}x\texttt{]} = (\sum \texttt{neighbor\_node[}x\texttt{]})/\texttt{number\_of\_neighbors} \quad (5)$$

$$\texttt{node\_data[}y\texttt{]} = (\sum \texttt{neighbor\_node[}y\texttt{]})/\texttt{number\_of\_neighbors} \quad (6)$$

Each node has two dimensional floating-point data and the computation functions are calculating the average values of all the neighbor nodes' data in two coordinate values respectively. Then the newly calculated data will be used for the next computation round. Each node needs access to its neighbor nodes' data to finish computing its position for the next step. After computation, the local processor will communicate with other processors to exchange border/ghost node information.

The locality management module experiment is done in two steps: First we test the performance of memory alignment in a static environment without any adaptation. Then we conduct our experiments in a dynamic environment where the system induces dynamical resource changes to simulate a real space-shared and time-shared runtime environment.

6.3.1 Computation Time in Static Environment

First we test the performance enhancement from memory alignment in a static environment, in which we use eight processors, 128 partitions and with processor relative weights {1,1,1,1,1,1,1,1}, which means we run the experiments on 8 processors and each processor provides the same time share for the experiments. In other words, the weight stands for computation time. For each experiment the number of iterations will be 1000, 5000, and 10000 respectively. All data is in seconds.

Graphs/ Iterations		**1000**		**5000**		**10000**	
Name	|V|, |E|	Time	speedup	Time	speedup	Time	speedup
wing	|V| =62032	26.95	29.3%	125.6	22.7%	257.0	29.7%
	|E| = 121544	19.04		97.04		180.60	
brack2	|V| =62631	24.35	13.2%	121.58	11.7%	245.12	12.2%
	|E| = 366559	21.13		107.41		215.30	
wave	|V| =156317	62.84	14.3%	319.52	15.9%	653.89	16.4%
	|E| = 1059331	53.87		268.80		546.60	

Table 9 Computation time in a static environment

Table 9 illustrates the test result, where the shaded cells contain normal computation time without memory alignment, and the non-shaded cells are computation time with memory alignment. All numbers are in seconds. In this table we can see that our memory management successfully shortens computation time by 11.7% to 29.7%. Different iteration time also shows that the enhancement is steady and the module has no slow-down when the graph's size becomes larger. Figure 19 also shows these improvements in separate charts.

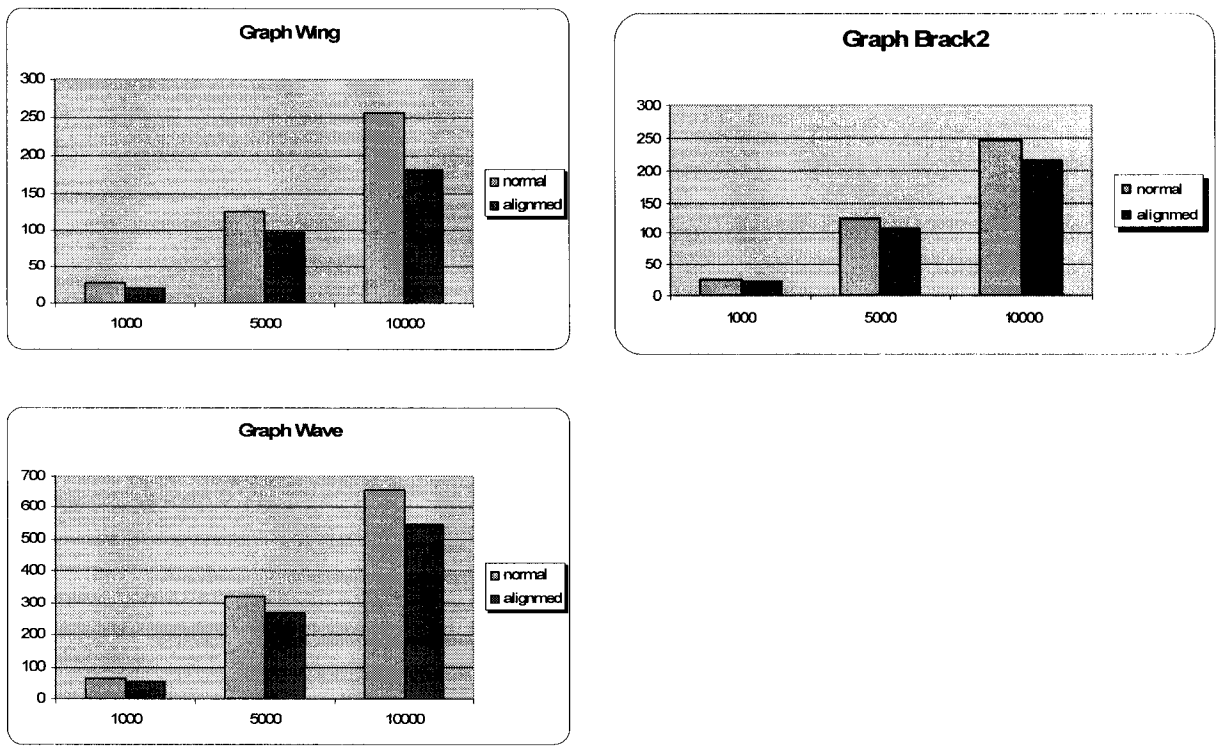

Figure 19 Computation time charts in a static environment

6.3.2 Computation Time in Dynamical Environment

In order to test our system in a dynamic runtime environment, we execute our experiments on 16 processors with 128 partitions. Initially we set each processor with the same computation weights and all processors take part in computing. Then the adaptations follow the processors weight matrix below, in which the first row stands for the initial status. In this matrix, we intentionally set up mixed time-shared and space-shared environment, see the matrix below. The first word in each row stands for which algorithm system us to make adaptation. If it is **atop**, then adaptation will be done by using ATOP over-partitioning algorithm; if it is **zoltan**, the adaptation will be done by using a partitioning algorithm from the Zoltan library to partition from scratch. K-way partitioning algorithm from the Zoltan library is used for our experiments. The numbers in each row stand for the computation weight on each processor, which means the computation time on each processor. The first two rows simulate a time-shared environment, and the last two rows simulate a space-shared environment. We use the **zoltan** directly in the third row to have the system made partitioning from scratch by using Zoltan library. Before each adaptation, the application repeats 1500 times of computation. Similarly, three graphs with different size are chosen to conduct the experiment.

{atop, 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}

{atop, 1,1,1,1,1,1,1,1,2,2,2,2,2,2,2,2}

{zoltan, 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}

{atop, 1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0}

{atop, 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}

Table 10 shows our experimental results. In this table, we also see the time used for memory alignment, which is the total time of five alignments after the five adaptations. Compared to computation time, the alignment cost is ignorable but the system speedups for experiments on different graphs are obvious. When we calculate the speedups, we have already counted in the memory alignment cost. So we can get speedup by 16.1% to 21.7% on these graphs after applying our new data locality management.

	wing		**brack2**		**wave**	
	Alignment	**Computation**	**Alignment**	**Computation**	**Alignment**	**Computation**
Non-aligned	0.0	92.68	0.0	98.5	0.0	255.6
Aligned	0.08	74.85	0.11	77.05	0.37	214.1
Speedup	19.15%		21.7%		16.1%	

Table 10 Computation time in a dynamic environment

6.4 Experiment Summary

By using the same set of graphs used in [20] we clearly illustrate that our new implementation with optimized data structure get much better adaptation performance among various graphs. Initial setup time can reduce by 99% and adaptation can also be improved by 55% to 97%. The main reason for the improvement is that in the original ATOP implementation, finding a node needs **O**(n) operation but in our implementation only **O**(0) operation is needed. The adaptation time is related to how much data needs to be moved. As long as the graph is getting larger, the original implementation leads to much larger turnover; our implementation, however, doesn't need search node in an array one by one for processing migration, which makes the system more scalable and improves for large graphs. It also make sense that we get lower performance enhancement in a time-shared environment compared to the improvements in a space-shared environment, because in time-shared environment the system is expected to migrate less data to finish an adaptation.

On the other hand, other factors such as the number of edges and the patterns of edges also play a role for how efficiently an adaptation can be done. First, these factors affect the adaptation performance by affecting the time of partitioning. Second, they also can affect the quality of partitioning. In other word, these factors also affect how much data needs to be moved during adaptation process. This is why we can not conclude the final performance by just the number of vertex and the number of edges.

Our experiments also show that by applying the memory alignment strategy, our data locality management module improves system performance by up to 29.7% in a static environment and by up to 21.7% in a dynamic environment. Also the speedups are not related to the size of the graphs.

7 Further Possible Improvement

7.1 Optimized Data Structure

The current data structure works well; however, it also shows some weakness during our implementation and experiments: border list objects are managed at mesh level which is too coarse-grained because we only have the number of processors border lists in total on each local mesh. In addition, communication is handled partition-based, so both node and partition need some effort to track the corresponding border data in the border lists, which makes the three objects ***mesh, partition*** and ***node*** somehow coupled together when dealing with border and ghost cell management. For example, each node needs four pointers to handle the border information, which makes the system not scalable well for some huge data graphs. In order to overcome this problem, we redesigned our architecture as shown in Figure 20, which mainly optimized the border and ghost management by putting the border list under the partition level. The ghost list is no longer a partition but an independent type, which makes the mapping between border list and ghost list easier and the system better maintainable. More important, the node structure becomes smaller and decoupled from how border list is handled.

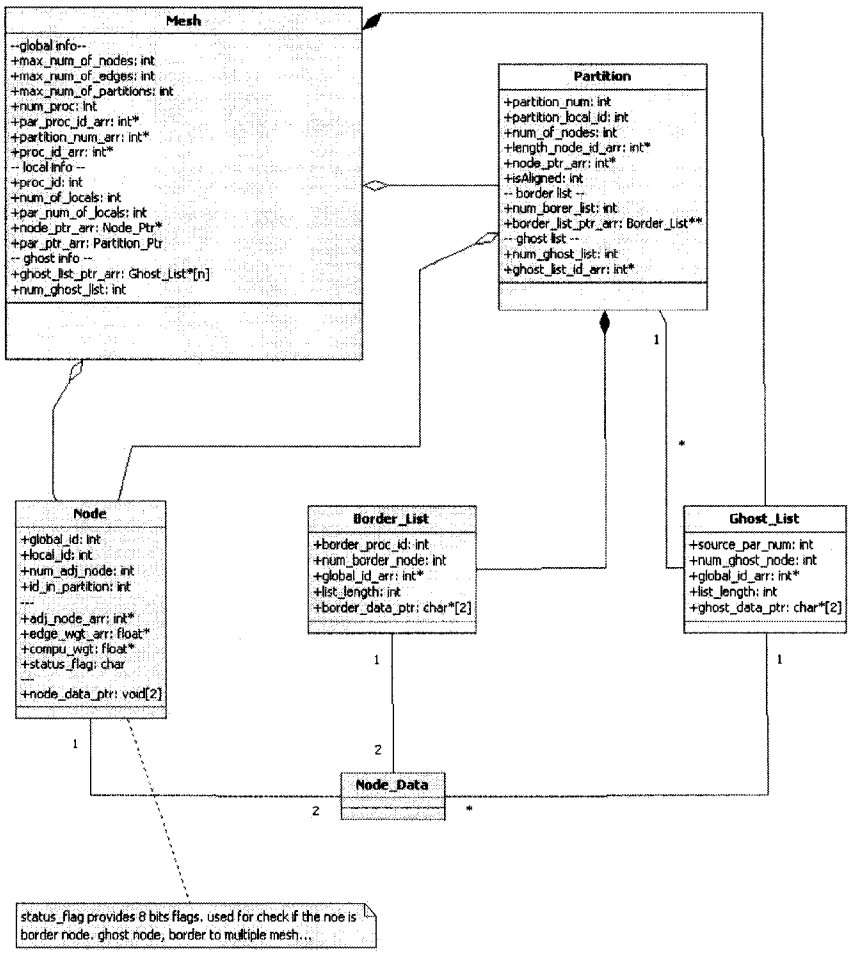

Figure 20 Optimized Class Diagram

7.2 Adding Memory Management

Another weakness is lacking further memory management. In the current system ***malloc()*** from C programming language is used to allocate memory dynamically. The system call ***malloc()*** uses a kind of linear search to find a proper size of memory from the heap for each memory allocation. This is suitable for large size memory allocation when each allocation is used for long term. In a typical particle simulation application, all initializing of nodes, packing nodes for migration, and unpacking nodes from migration

needs to dynamically allocate and de-allocate small size of memory. Since there are a great number of nodes in a simulation, those small-size memory chunk allocations and de-allocations will take a lot of time and affect system performance.

Our optimized memory management module design will make memory allocations at two levels: block level and node level. The block level algorithm allocates and frees large-size memory blocks, and tries to eliminate memory fragmentations. The node level allocation algorithm allocates small-size memory chunks without search and can reuse memory.

7.2.1 Block Level Management

The block level management uses a linked list of memory holes/blocks, which are **ordered by memory address**, called free list. Each hole is multiple times as big as a BLOCKSIZE, which can be defined at compile time for performance tuning purpose. Every time when we need to allocate a chunk of memory, we round the chunk size to multiple times of the BLOCKSIZE, and find a proper size of block from the linked list. If we cannot find a fitting one, we will allocate a new big memory chunk, say, 100 * BLOCKSIZE, and add it to our linked list.

```
struct block
{
   block *next;
   void *freelist;
   int size;
   short nodeSize;
   short allocated;
};
```

Figure 21 Block Data Structure

See the Figure 22; shaded parts are available free blocks in the linked list. Un-shaded parts are used by the application. They are not in the linked list. When allocating memory, the system always finds the first block/hole that is bigger than the memory size needed, then rounds the needed size to multiple times of BLOCKSIZE and allocate it. If the free block is larger than needed, the rest of it will be put back to linked free list.

```
# define BLOCKSIZE 512
Linked list:   struct block * holes;
```

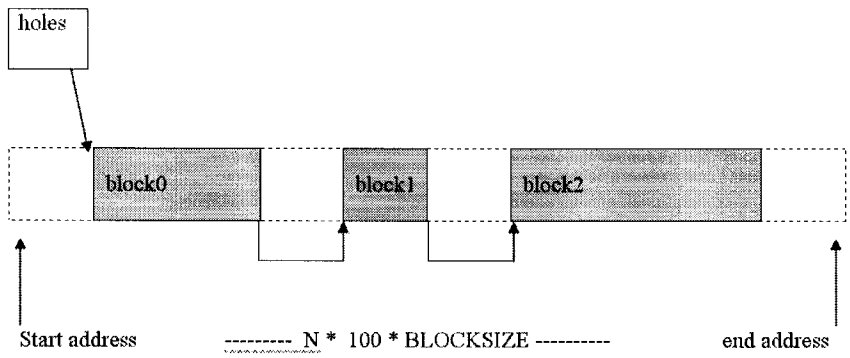

Figure 22 Linked List of Block before Allocation

For case 1, assume BLOCKSIZE = 512, we need to allocate a memory chunk with size 100. Then after we allocate it the list like Figure 23.

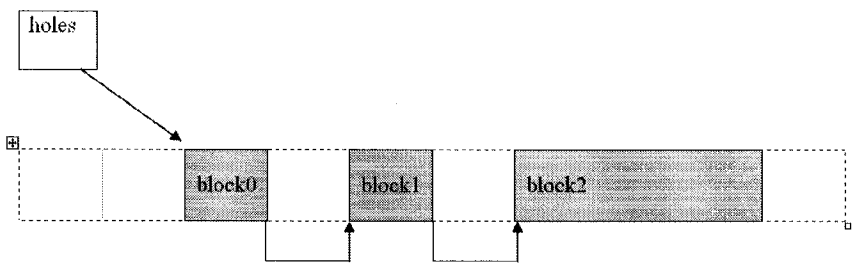

Figure 23 Linked List of Block after Allocation

For case 2, assume a block of memory, say A, is to be freed, then the linked list can be shown as Figure 24 and Figure 25.

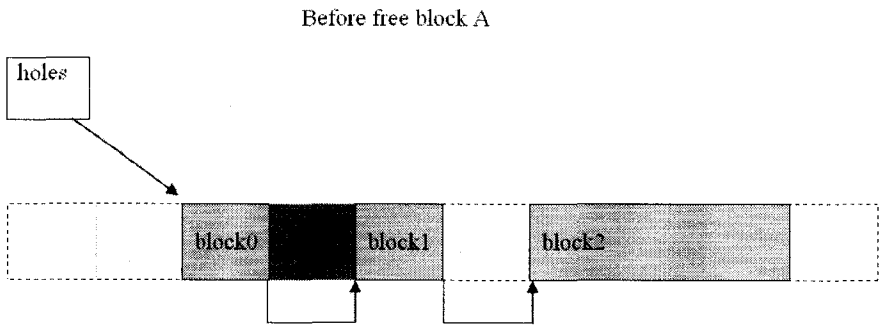

Figure 24 Before free block A

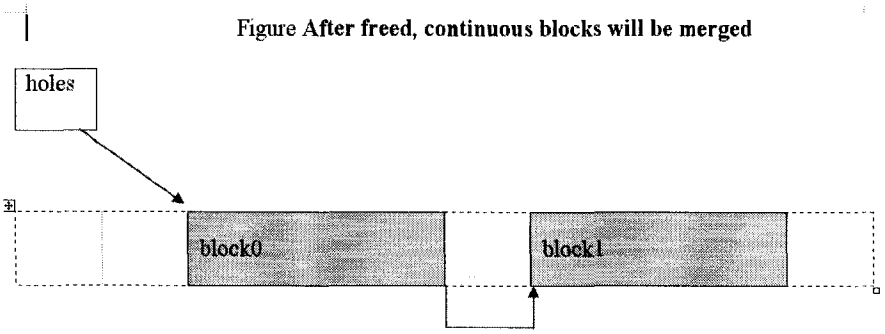

Figure 25 After free block A

7.2.2 Node Level Management

Compared to the block, the node size is much smaller than BLOCKSIZE. Each block can hold multiple nodes and each block only holds nodes that have the same size; and keeps a linked list for the same node-size blocks by using a LIFO algorithm. This way, each block can use a linked list to link the free slots in the block. When de-allocating a block, the block's memory is free and added back to a specific memory block. Thus helps to recycle memory blocks without fragmentation. The de-allocated space will be the first available node-sized memory for next allocation. Each block keeps a linked list to available memory, see Figure 26.

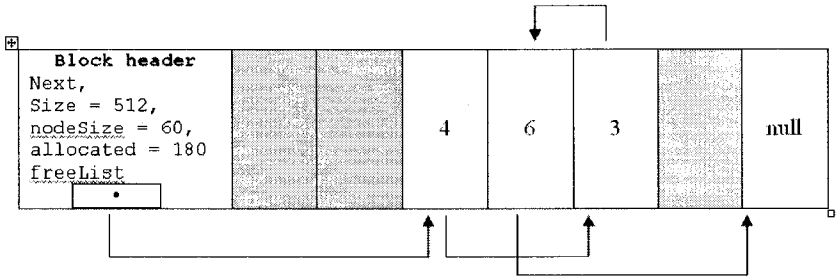

Figure 26 Internal Block Structure

An array of pointers to different node-size memory blocks is shown in Figure 27. Once we know the size of the node, we can always find the corresponding array index, and then get to the block which has available memory space for the node. Here another linked list is kept for the specific node-size blocks which are neither empty (ones that should be put back to the block-level list) nor full (ones that don't need record).

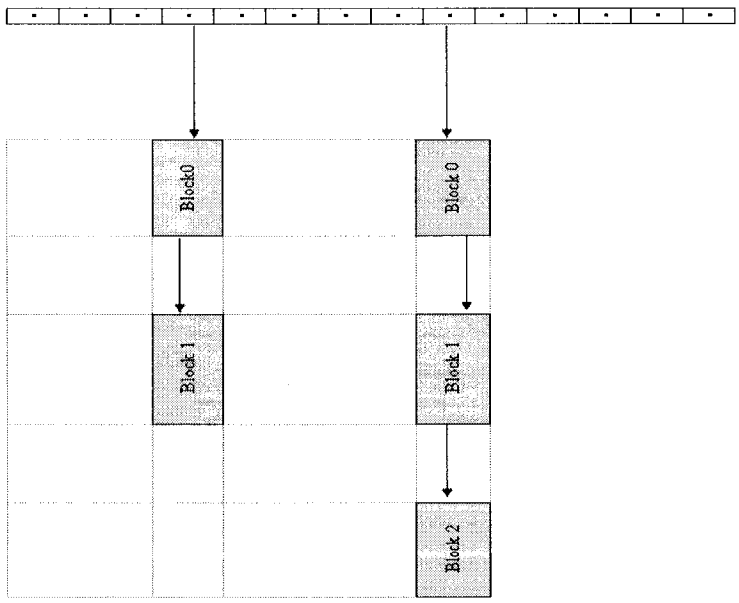

Figure 27 Blocks for Different-sized Nodes

7.3 Latency Hiding by Asynchronous Communication

It is possible to hide communication latency by using asynchronous communication. In our system this is managed in function **ATOP_MESH_CMMPUTE().** Appendix D shows the current implementation but it is not a fully asynchronous algorithm. After each round of computation, the system needs communication among processors to exchange border and ghost information. The idea is that it is not necessary to make the communication for all nodes synchronously after each round of computation.

In order to hide the communication latency, we initiate the communication phase to send out border data by using a non-blocking I/O API after each computation for a partition. After having finished the computation for one partition, updated border-data information will be sent out to corresponding processors by using border-list information stored in the current partition. After having finished one round of computation for the last local partition, the first finished local partition may have its updated ghost node data ready for the next round of computation. Pseudo code for showing the algorithm is shown in Figure 28 and more details are explained below:

1) The local mesh prepares for asynchronously receiving its ghost data for current round by using MPI_Irecv() for each partition. Current-round ghost-cell data will be used for computation in the next round. Since we are sending out border data in the next step based on each partition, the receiving will be finished for each partition separately.

2) **For each partition** in the local mesh, the system switches ghost data to the last round first. Then it checks whether all asynchronous receiving function calls for the partition in the last round have been finished. This is done by using MPI_WAIT(). If all last-round receiving calls for the partition have finished, all ghost-cell data of this partition is ready and the system starts to do current round computation. After finishing the computation for the partition, all border data of this partition will be send out asynchronously for the current round by using MPI_Issend(), which are asynchronously received in the step 1).

3) After having finished computation, the system swaps the values of variable **current_round** and variable **last_round** to start next computation round.

```
int current_round = 1;  // index, current round of computation
int last_round = 0;     // index, last round of computation
int send_tag;
int recv_tag;

for (int i = 0; i < max_round; i++){
   1. if (i != 0){
        for each ghost_list in current mesh:
        //asynchronously receive node data in ghost list for
      the
        //next round.
            recv_tag = dotage(source_partition_num,
                        local_processor_num, current_round);
            MPI_Irecv(…,ghost_data_ptr[current_round],
                              recv_tag[current_round], …);

      }

   2. for each local partition:
      1) if (i!= 0){
            ghost_list_id_arr[last_round]
         }

      2) for each related ghost_list for current partition
            MPI_WAIT(tag[last_round]...)
         end for each

         for each node in the partition:

                  node->node_data[current_round]
                  = compute (node->node_data[last_round]);
         end for each

      3) send out updated node data from current round of
         compuatation.
            tag = dotage(current_partition_num,
                        target_processor_num,current_round);
            MPI_Issend(…, border_data_ptr[current_round],
                              send_tag[current_round], ...)

     end for each of 2.

   Swap value of current_round and last_round
}//end of for
```

Figure 28 Latency Hiding Algorithm

8 Discussion and Conclusion

We have fully implemented a middleware for particle-simulation applications which can dynamically allocate resources in a time shared and space shared environment. By re-designing and implementing the ATOP algorithm using optimizing data structures and adding two levels of memory management, the library now is more realistic and re-usable for scientific particle simulations. Also by providing a higher-level interface to hide lower-level implementation details, users can spend more time on simulations instead of taking care of resource management and data migration management. Our experiments show that our library has improved the adaptation time by 55-99 % and has about 20% speedup from computation time by applying our data locality management to benchmark graphs.

Chapter 6 provides some useful design approach to further optimize our runtime library. We explained our new optimized data structures and memory management strategy. Thus, a potential future project from our AlphaMeta group will provide the new optimized implementation for this library, and integrate the middleware with job scheduler and resource monitor to make the whole library more useable for real users.

APPENDIX A

Bibliography

[1] RD Blumofe, CF Joerg, BC Kuszmaul, CE Leiserson, *Cilk: an efficient multithreaded runtime system* Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming table of contents, Santa Barbara, California, United States,Pages: 207 - 216 Year of Publication: 1995 ISSN:0362-1340

[2] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, W.F. Mitchell, M.S. John, and C. Vaughan. *Zoltan—Data-Management Services for Parallel Applications User's Guide*. Sandia National Laboratories, 2002

[3] K. Devine, B. Hendrickson, E. Boman, M.St. John, and C. Vaughan. *Design of dynamic load-balancing tools for parallel applications.* In Proc. of the Intl. Conference on Supercomputing, 2000

[4] P. Diniz, S, Plimpton, B. Hendrickson, and R Leland. *Parallel algorithms for dynamically partitioning unstructured grids*. Proc. 7th SIAM Conf. Parallel Proc., 1995

[5] J. Flaherty, R. Loy, C. Ozturan, M. Shephard B. Szymanski, J. Teresco, and L. Ziantz. *Parallel structures and dynamic load balancing for adaptive finite element computation.* Appl. Numer. Maths, 26:241–263, 1998

[6] Bruce Hendrickson and Robert Leland. *A multilevel algorithm for partitioning graphs*. Technical Report SAND93-1301, Sandia National laboratories, 1993

[7] Kai Hwang, Zhiwei Xu *Scalable Parallel Computing* page 237 published by WCB McGraw-Hill 1998 ISBN 0-07-031798-4

[8] Chao Huang, Orion Lawlor, L. V. Kale *Adaptive MPI* Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03) 2003

[9] G. Karypis and V. Kumar. *Multilevel k-way partitioning scheme for irregular graphs*. Technical Report TR95-064, Department of Computer Science, University of Minnesota, 1995

[10] G. Karypis, K. Schloegel, and V. Kumar. PARMETIS—*Parallel Graph Partitioning and Sparse Matrix Ordering Library version 2.0*, September 1998.

[11] Girija J. Narlikar and Guy E. Blelloch *Space-Efficient Implementation of Nested Parallelism* June 1997 ACM SIGPLAN Notices , Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming, Volume 32 Issue 7

[12] L. Oliker and R. Biswas. PLUM: *Parallel load balancing for adaptive unstructured meshes.* Journal of Parallel and Distributed Computing, 52(2):150–177, 1998

[13] C. Ou and S. Ranka. *Parallel incremental graph partitioning using linear programming*. Proceedings Supercomputing '94, pages 458–467, 1994.

[14] C. Ou, S. Ranka, and G. Fox. *Fast and parallel mapping algorithms for irregular and adaptive problems*. Journal of Supercomputing, 10:119–140, 1996

[15] A. Patra and D. Kim. *Efficient mesh partitioning for adaptive hp finite element meshes.* Technical report, Dept. of Mech. Engr., SUNY at Buffalo, 1999

[16] J. Pilkington and S. Baden. *Dynamic partitioning of non-uniform structured*

workloads with space filling curves. Technical report, Dept. of Computer Science and Engineering, Univ. of California, 1995

[17] K. Schloegel, G. Karypis, and V. Kumar. *Multilevel diffusion schemes for repartitioning of adaptive meshes*. Journal of Parallel and Distributed Computing, 47(2):109–124, 1997

[18] K. Schloegel, G. Karypis, and V. Kumar. Wavefront diffusion and LMSR: *Algorithms for dynamic repartitioning of adaptive meshes.* Technical Report TR 98-034, University of Minnesota, Department of Computer Science and Engineering, 1998

[19] K.Shen, H.Tang, and T, Yang. *Adaptive two-level thread management for fast MPI execution on shared memory machines*. In Proc. ACM/IEEE Conf. on

[20] Angela C. Sodan, Lin Han, *ATOP-space and time adaptation for parallel and gird applications via flexible data partitioning*, Proceedings of the 3rd workshop on Adaptive and reflective middleware, Toronto, Ontario, Canada, Page: 268 – 276, 2004

[21] A. Sohn. S-HARP: *A parallel dynamic spectral partitioner*. Technical report, Dept. of Computer and Information Science, NJIT, 1997

[22] A. Sohn and H. Simon. JOVE*: A dynamic load balancing framework for adaptive computations on an SP-2 distributed memory multiprocessor*. Technical Report 94-60, Dept. of Computer and Information Science, NJIT, 1994

[23] R. VanDriessche and D. Roose. *Dynamic load balancing of iteratively refined grids by an enhanced spectral bisection algorithm*. Technical report, Dept. of

Computer Science, K. U. Leuven, 1995

[24] C. Walshaw, M. Cross, and M. G. Everett. *Parallel dynamic graph partitioning for adaptive unstructured meshes.* Journal of Parallel and Distributed Computing, 47(2):102–108, 1997

[25] M. Zaki and W. Li and S. Parthasarathy, *Customized Dynamic Load Balancing for a Network of Workstation,* Journal of Parallel and Distributed Computing, volume 43, number 2, pages 156-162, 1997

[26] The Graph Partitioning Archive, http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/

APPENDIX B

Data Structures

```
struct Mesh
{
      /**** global info variables ****/
      int max_num_of_nodes;   /* the number of nodes for global mesh */
      int max_num_of_edges;   /* the number of edges for global mesh*/
      int max_num_of_partitions;    /* NUM_PARTITION */
      int num_proc;                 /* number of processors */

      int* proc_id_arr;       /* global table for processor id of
                                 every node */
      int* partition_num_arr; /* global table for partition_num of
                                 every node.*/
      int* par_proc_id_arr;   /* global table for processor id of each
                                 partition */


      /**** local info variables ****/
      int proc_id;            /* processor id where the mesh locates */

      /** mapping table for local partition **/
      int par_num_of_locals;  /* number of partitions on this local
                                 mesh */

      Partition_Ptr* par_ptr_arr; /* array of pointers to partitions */

      /** mapping table for local nodes **/
      int  num_of_locals;     /* how many local nodes */
      Node_Ptr* node_ptr_arr;

      Ghost_List** ghost_list_ptr_arr;    /* array of pointer to
                                             ghost_list.*/

};

typedef struct Mesh Mesh;
typedef struct Mesh * Mesh_Ptr;
```

```
struct Partition{

      /* identity information for current partition */
      int partition_num;            //partition global id
      int partition_local_id;       //index of the partition in m

      /* node information of current partition*/
      int num_of_nodes;             // number of nodes in artition
      Node_Ptr* node_ptr_arr;       // an array of node_ptr


      /* if partition is located on a aligned memory, difference will
         affect partition destroy */
      int isAligned;

      /* border info */
      Border_List** border_list_ptr_arr;  /* array of pointer to a
                                             Border_List.*/
      int num_borer_list;     // num of processors the partition border
      int* border_proc_id;    // array of bordered processor id.


      /* ghost info */
      int num_ghost_list;     // number of related ghost list
      int* ghost_list_id_arr; /* array of ghost list id that current
                                 partition needs */

};
```

```
struct Node
{

      int global_id;
      int local_id; /* index of this node in local_global_map array */

      int num_adj_node;       /* number of neighbour of this node */
      int *adj_node_arr;      /* list of adjacent nodes */

      int   id_in_partition;

      float *edge_wgt_arr;    /* communicational workload or edge
                                 weight) for adjacent nodes */
      float *compu_wgt;       /* computational workload */

      char  status_flag:      /*bit-map flag variable for node status*/

      void* node_data_ptr[2]; /* pointers array point to node data. */

};

typedef struct Node Node;
typedef struct Node *Node_Ptr;
```

```
struct Border_List
{
      int border_proc_id;      /* border to which processor */
      int num_border_node;     /* number of node in the border list */
      int* global_id_arr;      /* array of border nodes' id in current
                                  partition */
      int list_length;         /* length for border list, for memory
                                  management */
      char* border_data_ptr[2];     /* pointer to a block of memory for
                                       border node data */

};


struct Ghost_List
{
      int source_par_num;      /* ghost list from which partition */
      int num_ghost_node;      /* number of node in the ghost list */
      int* global_id_arr;      /* array of ghost nodes' id from the
                                  ghost partition */
      int list_length;         /* length for the ghost list, for memory
                                  management */
      char* ghost_data_ptr[2];/* pointer to a block of memory for ghost
                                 node data */
};
```

APPENDIX C

Sample Application

```
#include <atop.h>

#define YES       1
#define NO        0

/*************************************************************
 *  STEP 1. define two call-back functions
 *  one is for the computing data size of each node.
 *  the other one is computing function for each node.
 *************************************************************/
int node_data_size()
{
      return sizeof(float) *2;
}


void calculator(ATOP_MESH_PTR atop_mesh_ptr, ATOP_NODE_PTR
atop_node_ptr)
{
      int i, nbor_global_id;
      float* data_arr;
      float* result_arr;

      float sum1 = 0, average1 = 0;
      float sum2 = 0, average2 = 0;

      ATOP_NODE_PTR temp_node_ptr;

      //printf("----------- entering calculator ----------------\n");
      for(i = 0; i < atop_node_ptr->num_adj_node; i++)
      {
            nbor_global_id = atop_node_ptr->adj_node_arr[i];
            temp_node_ptr = ATOP_GET_NODE(atop_mesh_ptr,
            nbor_global_id);

            data_arr =
            (float*)temp_node_ptr->node_data_ptr
                        [atop_mesh_ptr->old_data_index];

            sum1 += data_arr[0];
            sum2 += data_arr[1];
      }

      average1 = sum1/atop_node_ptr->num_adj_node;
      average2 = sum2/atop_node_ptr->num_adj_node;

      result_arr = (float*)atop_node_ptr->
                  node_data_ptr[atop_mesh_ptr->new_data_index];

      result_arr[0] = average1;
```

```
        result_arr[1] = average2;
}


int main(int argc, char **argv)
{
        int i,j;
        int max_num_partition;
        int adapt_loop = 4;
        int test_loop = 10000;

        float* data_arr1;
        float* data_arr2;

        int partition_weights_matrix[5][5]={
                                        {ATOP,1,1,1,1},
                                        {ZOLTAN,1,2,1,1},
                                        {ATOP,1,1,0,0},
                                        {ATOP,1,1,2,2},
                                        {ATOP,1,1,1,1}
                                };


        char* graph_name = "../Graphs/wave.graph";
                                /* graph file name */


        ATOP_NODE_PTR my_node_ptr;
        ATOP_MESH_PTR my_mesh_ptr;

        max_num_partition = 128;


        /*****************************************************
         *  STEP 2 register user defined callback functions */
        /*****************************************************/
        ATOP_Set_Data_Size_Fn(&node_data_size);
        ATOP_SET_MESH_COMPUTE_FN(&calculator);

        /*****************************************************/
        /*                 STEP 3 initialize mesh      */
        /*****************************************************/
        ATOP_ENVIRON_INIT(argc, argv);
        my_mesh_ptr = ATOP_CREATE_MESH(graph_name, max_num_partition);

        /* initialize node data for every node */
        for(i = 0; i <my_mesh_ptr->num_of_global_nodes; i++)
        {

                my_node_ptr = ATOP_GET_NODE(my_mesh_ptr, i);

                /* make sure only initialize data on the processor which
                owns the node */
                if(my_node_ptr != NULL)
                {
                        /* if we need float[2] for each node_data */
                        data_arr1 = (float*) malloc(sizeof(float) * 2);
```

```
                data_arr2 = (float*) malloc(sizeof(float) * 2);

                if(data_arr1 == NULL || data_arr2 == NULL)
                {
                printf("Can not allocate memory for node data!! \n");
                      exit(1);
                }

                data_arr1[0] = data_arr2[0] = i;
                data_arr1[1] = data_arr2[1] = i*2;

                my_node_ptr->node_data_ptr[0] = data_arr1;
                my_node_ptr->node_data_ptr[1] = data_arr2;

                //node_display((Node_Ptr)my_node_ptr);
          }
     }



     /*******************************************************/
     /*          STEP 4  First Adaptation                   */
     /*******************************************************/

     /*first time doing partition and migration. probably decided by
     users*/

     for ( i = 0; i < adapt_loop; i++){
          ATOP_ADAPTATION(my_mesh_ptr,  partition_weights_matrix[i%5]);

           //mesh_border_validate((Mesh_Ptr) my_mesh_ptr);
           //mesh_validate_ghost((Mesh_Ptr) my_mesh_ptr);
           //ATOP_MESH_DISPLAY(my_mesh_ptr);
     }


     /*******************************************************/
     /*                      STEP 5 COMPUTATION             */
     /*******************************************************/
     for ( i =0 ; i < test_loop; i++){
           ATOP_MESH_COMPUTE(my_mesh_ptr);

           /* this method decides to do atop partition or zoltan
                       partition and following migration*/
           ATOP_TRY_ADAPATION(my_mesh_ptr);
     }

     /*******************************************************/
     /*                      STEP 6  Finalize  */
     /*******************************************************/
     //ATOP_MESH_DISPLAY(my_mesh_ptr);
     ATOP_MESH_DESTROY(my_mesh_ptr);
     ATOP_ENVIRON_DESTROY();

     return 0;
}
```

APPENDIX D

ATOP_MESH_CMMPUTE()

```
/*********************************************************************
*
      function: ATOP_MESH_COMPUTE()
      input:            ATOP_MESH_PTR
      output:           void
*********************************************************************/
void ATOP_MESH_COMPUTE(ATOP_MESH_PTR atop_mesh_ptr)
{
      int i,j,index;
      Node_Ptr node_ptr;
      ATOP_NODE_PTR atop_node_ptr;
      Mesh_Ptr local_mesh_ptr;
      Partition_Ptr partition_ptr;
      int tag, par_proc_id;
      void* buffer;
      int buffer_size;
      int source_proc;
      int target_proc;

      MPI_Request *request;
      MPI_Status  *status;
      int counter = 0;
      int k = 0;


      local_mesh_ptr = (Mesh_Ptr) atop_mesh_ptr;

      request = (MPI_Request*) malloc (sizeof(MPI_Request) *
      local_mesh_ptr->max_num_of_partitions *2);
      status  = (MPI_Status*) malloc (sizeof(MPI_Status) *
      local_mesh_ptr->max_num_of_partitions *2);

      /* First, asynchronized to receive those income ghost info */
      for( i = 0; i < local_mesh_ptr->max_num_of_partitions; i++)
      {
            if(local_mesh_ptr->ghost_par_id_arr[i] == TRUE)
            {
                  /* need updated border data from this partition*/
                  par_proc_id = local_mesh_ptr->par_proc_id_arr[i];

                  tag = 1000* (meshPtr->proc_id +1) + i;
                  buffer = local_mesh_ptr->index_in_buffer_arr[i];
                  buffer_size = local_mesh_ptr->num_ghost_arr[i]
                                    *__node_data_sizeof();


                  MPI_Irecv(buffer, buffer_size, MPI_CHAR,
                              par_proc_id, tag, MPI_COMM_WORLD,
                              &request[counter++]);
```

```
            mesh_update_ghost(local_mesh_ptr);
      }
}


/* Do computing for each local partition*/
for (i = 0; i < local_mesh_ptr->par_num_of_locals; i++)
{
      /* take a local partition */
      partition_ptr = local_mesh_ptr->
            par_ptr_arr[local_mesh_ptr->par_local_global_map[i]];

      for (j = 0; j < partition_ptr->num_of_nodes; j++)
      {
            node_ptr = partition_ptr->node_ptr_arr[j];
            atop_node_ptr = (ATOP_NODE_PTR)     node_ptr;
            __calculator(atop_mesh_ptr,atop_node_ptr);

            if(node_ptr->isBorderCell)
            {
                  mesh_update_border_data(node_ptr,
                                          local_mesh_ptr);
            }
      }

      /* update those ghost cells belongs to this partitions */
      for (j = 0; j < partition_ptr->num_proc; j++)
      {
            mesh_collect_border_data(partition_ptr);
            if(partition_ptr->bordered_proc_id_arr[j] == TRUE)
            {

                  /* this tag will be the same as that from
                                          receiving processor */

                  tag = (j+1) * 1000 +
                              partition_ptr->partition_num;
                  buffer = partition_ptr->start_index_arr[j];
                  buffer_size =
                        partition_ptr->num_bordered_node_arr[j] *
                        __node_data_sizeof();


                  MPI_Issend(buffer, buffer_size,
                              MPI_CHAR, j, tag,
                        MPI_COMM_WORLD, &request[counter++]);

            }
      }     // for loop j
}// for loop i

/*synchronize all request */
if (counter > 0){
      //MPI_Waitall(counter, request, status);
      for(i = 0; i < counter; i++)
      {
            MPI_Wait(&request[i], &status[i]);
```

```
            }
        }

        /*after finish a round of computation, swap new data index and
                                                old_data_index */
        mesh_data_index_swap(local_mesh_ptr);

        free(request);
        free(status);
}
```

APPENDIX E

Vita Auctoris

Name: Yu Zou

Place of Birth: Da Zhu County, P.R. China,

Education: 2007 M.Sc., Computer Science
University of Windsor
Ontario, Canada

2003 B.Sc., Computer Science
University of Windsor
Ontario, Canada

1995 M.A. Economics
Southwestern University of Finance and Economics
Chengdu, China

1989 B.E.Eng
SouthEast University
Nanjing, China