3-23-2009

An Adaptive Mesh MPI Framework for Iterative C++ Programs

Karunamuni Charuka Silva

AN ADAPTIVE MESH MPI FRAMEWORK FOR ITERATIVE C++ PROGRAMS

by

CHARUKA SILVA

Under the Direction of Sushil K. Prasad

ABSTRACT

Computational Science and Engineering (CSE) applications often exhibit the pattern of adaptive mesh applications. Adaptive mesh algorithm starts with a coarse base-level grid structure covering entire computational domain. As the computation intensified, individual grid points are tagged for refinement. Such tagged grid points are dynamically overlayed with finer grid points. Similarly if the level of refinement in a cell is greater than required, all such regions are replaced with coarser grids. These refinements proceed recursively. We have developed an object-oriented framework enabling time-stepped adaptive mesh application developers to convert their sequential applications to MPI applications in few easy steps. We present in this thesis our positive experience converting such application using our framework. In addition to the MPI support, framework does the grid expansion/contraction and load balancing making the application developer's life easier.

INDEX WORDS: Adaptive mesh, Parallelization, Time-stepped, Battlefield management simulation, Space Filling Curve (SFC), Object-oriented

AN ADAPTIVE MESH MPI FRAMEWORK FOR ITERATIVE C++ PROGRAMS

by

CHARUKA SILVA

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

in the College of Arts and Sciences

Georgia State University

2009

AN ADAPTIVE MESH MPI FRAMEWORK FOR ITERATIVE C++ PROGRAMS

by

CHARUKA SILVA

Committee Chair: Sushil K. Prasad

Committee: Raj Sundarraman
Saeid Belkasim

Electronic Version Approved:

Office of Graduate Studies
College of Arts and Sciences
Georgia State University
May 2009

DEDICATION

To my mother,

Who took tremendous challenge to bring me up this far.

To my sister,

Who always stands next to me as my second mother.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Sushil K. Prasad, for his expert guidance. He is always open to new ideas and eager to explore options.

Also I would like to thank Dr. Raj Sundarraman, Dr. Raheem Beyah and subsequently Dr. Saeid Belkasim for reviewing my thesis and putting valuable inputs.

I thank my project group members Chad, Srilaxmi, Akshaye, Sunetri, Sara and Joseph for extending their helpful hands in many ways to make my research work as well as my time in Georgia State University a success.

Also I thank my husband and my family for all their support and encouragements.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

In the field of Computational Science and Engineering many applications exhibit irregular structure and dynamic behavior with dynamic load patterns. One such application domain can be recognized as using Adaptive Mesh Refinement techniques (AMR). Basic algorithm of AMR technique is to begin with coarser grain mesh structure covering the entire computational area and later refine the particular grid points where interesting physical phenomena occur. It could be replacing the coarser grain with finer grain grid points or the other way round. This refinement proceed recursively when program progresses. Due to refinement at arbitrary points, mesh will eventually result in as an imbalance system. Sequential implementations of conventional adaptive mesh applications are very complex and difficult to validate. The parallel applications of those are far more complex than those. Therefore providing an infrastructure to such applications is clearly a benefit to the application developers. In this thesis we propose an object-oriented framework which takes few parameters to convert C++ adaptive mesh application into MPI application. This work follows our previous work iC2MPI [18] which converts graph structured –time stepped sequential C programs into MPI applications.

Many time-stepped applications such as fluid dynamics [6] or mesh structured applications such as difference equations [9], finite element methods [7], and cellular automata [21], incompressible flows [23], low Mach number models for atmospheric flow [24] and combustion [25] will definitely benefit from our framework as it handles the bulk of extra work required by programmers in parallelization of those applications.

1.1 Goal of the Framework

1. Design an open architecture framework for class of mesh structured C++ applications, which may require adaptive mesh refinement and dynamic load balancing.
2. Enable programmers to easily plug-in application code to the framework with minor changes to the original code to convert sequential C++ code into MPI based distribute applications and add the adaptive mesh refinement behavior to applications, which requires the same.
3. Enable developers performance tuning of the computation, communication and load balancing of the framework itself to effect in overall porting applications benefit

Our resultant framework is an improvement to our previous work iC2mpi [2]. Compared to any other AMR framework exist, we have provided quite easy conversion from sequential C++ AMR based application into MPI application and at the same time, if not already in-built, to support the AMR behavior to applications which require that, fulfilling our first and second goals. We have proved our concept porting sample applications with few code changes to the original application. Experiment results section exemplifies the speed up achieved (upto 12 speed up with 16 procs for 256 x 256 mesh) and error rate drop for quality achievement. The open architecture of our design provides ample room for interested developers to improve our framework from many aspects concentrating single area.

1.2 Section Breakdown

This thesis is organized as follows. Section 2 contains background information required for implementation of the framework. There we try to present the basic idea of space-filling curve and

expandable hash structure. Section 3 describes the related work. Section 4 presents the architecture of the framework and flow control. There we describe the application user input and points of inputs to the framework. In Section 4 we explain the design and implementation of our framework in detail. Section 5 describes our Experimental results while Section 6 summarizes our work and exposes the future improvements to the framework.

2. RELATED WORK

In this section we will discuss existing literature on adaptive mesh refinement frameworks and similar research work. The work of [22] investigates the use of spatial keys to uniquely identify and order the objects defining a triangular mesh. They do the triangular refinement using newest-node bisection. When refining they divide the mesh integral distance by four. They replace a line with a new node and two new lines. And bisect a triangle, replacing it with a new line and two new triangles. However their load balancing and communication between processors are not clearly described.

Rendleman et al., present [20] parallelization of block-structured adaptive refinement algorithms. Their work is supported by the software infrastructure provided by BoxLib library of [11]. BoxLib is a C++ library, which can be used for adaptive mesh refinement applications. BoxLib includes the basic parallelization facility hiding the detailed information from application developer. BoxLib creates the mesh structure required for the application and it handles the communication via MPI calls. Charles and team describe their implementation as having five components: error estimation and re-gridding routine to mark the refinement nodes, grid management routine manages the grid hierarchy, interpolation routines initializes a solution on newly created fine grids and interpolate the boundary conditions, synchronization routines correct mismatches at coarse/fine boundaries and integration routine distribute the grids among physical processors. The paper describes refinement algorithm and load balancing algorithm in detail. The differences in their work and ours are ease of porting, fairly large problem domain we address and control that a user has on his application on the framework.

Chien et al., focus on large scale shared memory systems for writing SMAR methods [14]. Their runtime supports the programs to utilize varying degree of shared memory support. Their implementation is done on top of Illinois Concert System and C++. Several threads employ in single address space. Shared memory is used for all communication and sharing.

Teng et al., propose to use balanced quadtrees and octrees to represent well shaped meshes[13]. Quadtrees and octrees can grow dynamically and adaptively to approximate the process of adaptive refinement of unstructured mesh. They focus on reducing dynamic load balancing to static partitioning and reducing parallel mesh refinement to a collection of traditional mesh refinements. Scheme first builds a balanced 2d tree to model the unstructured mesh. And they assume at refinement mesh will be refine at every region. Also they do not consider coarsening. The work has not presented their experimental results.

PARAMESH [26] is a package of Fortran90 implementation of Adaptive Mesh Refinement framework developed in NASA. As with our framework, goals of the PARMESH are to enable application developers to convert their serial code into parallel code and enable the regular mesh structured framework, the adaptive mesh behavior. The package builds a hierarchy of sub grids covering the entire computational domain. The sub grid blocks are organized as quad tree (for 2D) or oct-tree (for 3D). Each grid block has a logical Cartesian mesh. In this framework one of the restriction imposed on refinement is that a refinement level cannot jump by more than one refinement level at any location in the spatial domain. After every refinement stage, framework re-organizes the grid distribution among processors. At refinement, each block maintains the initial

mesh structure. They use Morton Space filling curve. Communication between processors is supported by MPI standard

PETSc [30] is a suite implemented in Mathematics and Computer Science Division of Argonne National Laboratory to produce numerical solution of partial differential equations and related problems on high-performance computers. PETSc use MPI standard for message passing communication as well. PETSc includes parallel linear, nonlinear equation solvers and time integrators for applications developed in C, C++ and Fortran.

Bhandarkar et al., focus on load balancing in MPI programs in [17]. They use the charm++ in their work to bring the dynamic load balancing capability to MPI programs. Charm++ [29] is an object-oriented parallel programming language. It gives the dynamic load balancing capabilities using runtime measurements of computational loads and communication patterns. Then it employs object migration to achieve load balance. Therefore any MPI program should be able to get the benefit of charm++ load balancing capability by simply transforming the MPI code into charm++ code. But conversion from MPI to charm++ requires the understanding of charm++ language. They propose a framework to do the conversion job.

Libmesh [32] library and toolkit provide underling adaptive mesh refinement capability for numerical simulation of partial differential equations. Currently it supports 1D, 2D, and 3D steady and transient finite element simulations. It makes use of existing software for solvers.

Chombo [35] provides software infrastructure for finite difference methods for the solution of partial differential equations on block-structured adaptively refined rectangular grids. Framework makes use of the previous work of work BoxLib [11]. It addresses both elliptic and time-dependent modules. The tools incorporated the visualization tool ChomboVis to visualize the data sets. MPI is used for communication in Chombo framework.

AMR framework [34] is designed for predicting physically complex flows. Framework is limited to 1 D and 2 D flows. It provides the basis for development of computational analysis tools for complex flow prediction. Block based adaptive mesh refinement is done and parallel implicit time stepping approach is applied.

Parashar et al., in their work [19] present data structures for adaptive mesh refinement. Two basic data structures are presented here: a Scalable Distributed Dynamic Grid, which is a single grid in an adaptive grid hierarchy and a Distributed Adaptive Grid Hierarchy, which is a dynamic collection of SDDGs. Computational data associated with the grids in the hierarchy is maintained as a scalable distributed dynamic array. Grid points are ordered according to the space-filling curve. The work shows experimental results with a representative application from numerical relativity and proves that the presented data structure has no significant overhead.

FEM [31] is an automatic load balancing framework developed for the Converse [31] interoperable runtime system. Framework records the computation time taken for each virtual process or char defined in charm++ [29] and communication end points for each virtual processor. With this information it creates the communication graph in each physical processor. It

implements a mechanism for object migration and plug in for load balancing strategy. The limitation of the framework is that it is designed for specific domain.

Autopilot [27] presents ways of collecting information at runtime for parallel programs. Also it provides fuzzy logic based decision engine to aid resource management in parallel programs. But it is left to the programmer to implement the decision provided by the fuzzy logic. Since the runtime system of the parallel program does not actively carry out the decision, the load balancing is not transparent to the parallel program. In our framework we manage the load balancing on behalf of the application developer.

Similarly system CARMI [28] informs the parallel program the load imbalance and leaves the load balancing process to be implemented by the application developer.

Aluru, Fatih [33] describe load balancing technique based on space filling curve. They present algorithms to linearly order points in multi dimensional space using the Z-curve and Graycode curve. Since the proximity preserving nature of the space filling curve, from multi dimensional space to one dimensional space, they are used for partitioning.

J¨orn Behrens and Jens Zimmermann [5] introduce new recursive space-filling curve algorithm for adaptively refined triangular meshes for their dynamic distribution. Their experiment results show good load balancing and edge cut characteristics.

3. FRAMEWORK ARCHITECTURE

This framework is designed to have a layered architecture to decouple the user application and the framework. Application user does not require having knowledge on node partition, distribution, load balancing and MPI calls. Application sits on top of the framework. Figure 1 shows the layered architecture of the platform. Application provides the underlying mesh structure, which is essentially the width of the mesh (number of grid points in each direction), node data structure, which can be a class implementation and node computation function, which may define the criteria for expansion and contraction of mesh, as user plug-ins to the framework. Platform uses a space-filling curve to convert n-dimensional mesh into 1-dimensional list, which will then partition among the processors. Also platform supports the adaptive behavior of the application carrying out mesh expansion and contraction when required. At the same time framework will take care dynamic load balancing. Load balancing algorithm can be replaced not harming to the existing framework functionality. The platform uses an MPI [12] approach for parallelization, one of the most widely used methods to achieve parallelism on today's clusters and multiprocessor supercomputers.

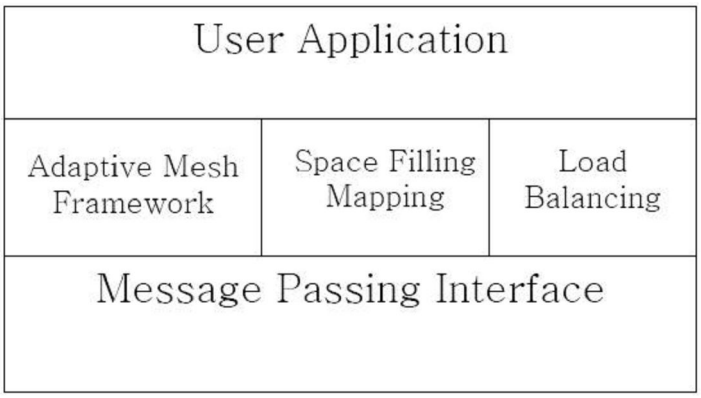


Figure 1: Layered architecture of the framework

The Figure 2 shows the detailed architecture of the framework and it explicitly shows the user plug-in points (underlying mesh size, node data structures and node computation function). White boxes in the figure show the user plug-ins to the framework and dark gray boxes represent the framework while light gray boxes shows the external applications and libraries used in the framework.

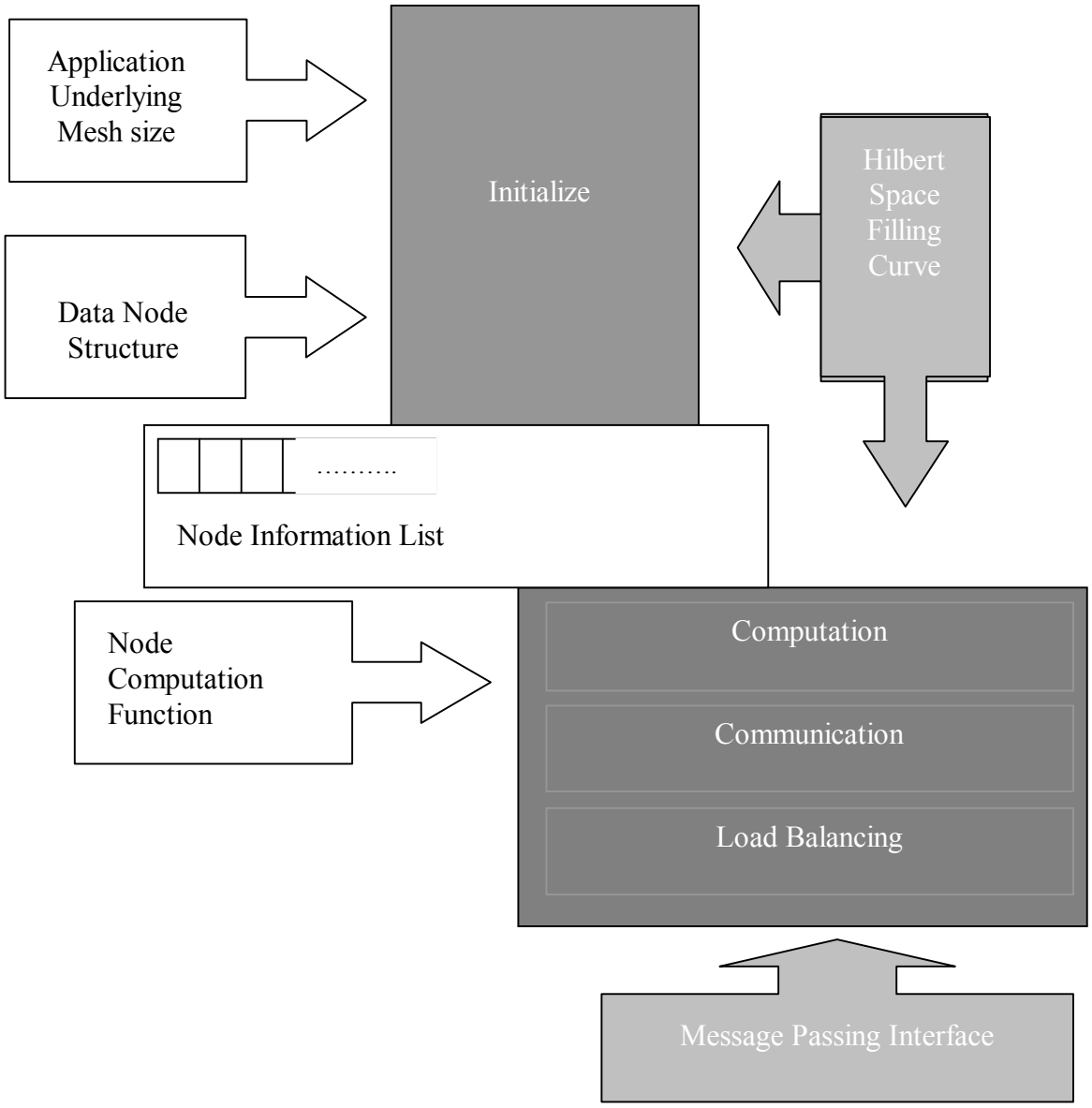


Figure 2: Detailed architecture of the framework

Application provides following inputs to the framework to interact with it.

1. Application mesh size. And degree (two dimensional or three dimensional) of the mesh handling by setting a variable in the framework.
2. Computational node structure definition, which can be a C++ structure or class and its implementation.
3. Computational function defined on each node. This function accepts pointer to grid point object on which the computation function is called, and a pointer to hash table of computational data list. Function defines the error function for node refinement and coarsening.

The application use the third party 'Hilbert's Space-Filling Curve' to get the one dimensional space filling curve generated from user application mesh. Framework will partition the generated one dimensional node list among the processors and create node-to-processor mapping, which will be used later to find out the shadow nodes for a processor as well as for load balancing. The second user input, the structure of computational data node of the application is recognized by the initialize phase of the framework to generate data nodes hash tables associated with the application graph. At the end of initialization phase all required data structures are created for next phase.

Third user input, "application computation function" is supplied during the computation and communication phase of the framework. Framework will call the computation function on each node and transport the communication data between relevant processors. New nodes may create

or destroy as a result of computation. Nodes requiring coarsening or refinement will be marked as a result of error function calculations and framework will do the actual node generation and coarsening according to the flag. Also location assignment (new mesh generation) neighbor nodes update and load balancing caused due to adaptive mesh refinement will be handled by the framework. So the framework essentially saves the programmer from doing all the background work and let him concentrate on the business logic. At the end of computation phase, framework supports communication between processors (packing, sending and receiving, updating information to prepare for next computation cycle.) If it is the load balancing cycle, framework does the re-partitioning and distribution of nodes between the processors without harming the program state.

3.1 Program Flow of Control

We recognize three main phases in the program as described below.

Initialization Phase

'Initialize Data Structures' uses the node data structures provided by the user. And set up the data structures for maintaining node information and node data in local memories of the processors. Node information contains the information about its neighbors and shadow node information to the processor. This phase could be a considerable overhead to large mesh applications. Since this overhead occurs only once in lifetime of an application, it should be bearable.

Computation and Communication Phase

'Compute over Nodes' function supplied from the framework initiate the computation of the framework. We pass a pointer to application specific, user given "computational function" as a parameter to the ComputeOverNodes function. Application Node function provided by the user incorporates the actual code for the node computation. Updated node data from the application node function is packed into communication buffers. The framework data structure keeps the neighbor information of each node, so that application developer can easily track the neighboring nodes for inter-grid computation and communication.

CommunicateShadows function calculates number of shadow nodes going to the neighboring processors. And pack the relevant data nodes and send them through MPI_Send. At the same time each processor prepare for receiving the updated shadow nodes from their neighbors. In every iteration framework needs to calculate the new buffer size for sending shadow nodes since there could be node splitting and contraction taken place during the computation phase. This buffer size calculation could be an extra overhead introduced by the framework. If we can keep some information on mesh configuration changes, we can control the buffer node creation. We are planning to consider this fact for further improvements on our framework. Computation and communication phases may be called several times as application flows.

Load Balancing Phase

The load balancing routine is periodically invoked. The load normalization is done in a centralized heuristic algorithm. Normalized load is calculated dividing the total number of nodes in the system by number of processors. We use the updated Space filling curve to re-partition the nodes among

processors. The information required to re-partition the nodes, will be gathered to processor 0. Then processor 0 will do the re-partition and inform everybody their new boundaries, so that each processor can eventually pack the outgoing nodes and send them to relevant processors or prepare to receive incoming new nodes from their neighboring processors.

Since the space-filling curve preserves the spatial locality, it is fair enough to consider the resulting distribution to be comparable with its traditional grid distribution in terms of communication overhead.

Figure 3 shows the flow control of the framework.

To run the application type ***mpirun –np <numberOfProcessors> Framework <meshSize>*** , Where **numberOfProcessors** is the number of processors the application is running and **meshSize** is the length of an edge of square or cubic mesh. To compile the application use the command *make all* and *make clean* will delete the objects files if any. Before each run output.txt should be deleted. This file stores the space filling curve order.

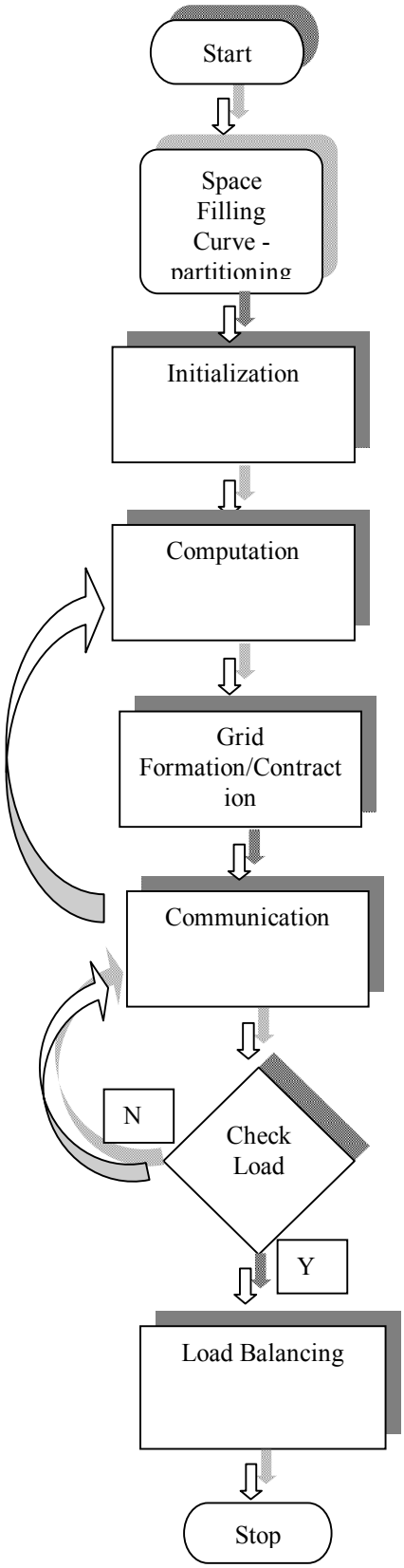


Figure 3: Control flow of the framework

4. DETAILED DATA STRUCTURES

Selection of data structure is vital to efficient computation of distributed systems. When designing the data structures we considered several factors: fast access, flexibility and ability to accommodate mesh expansion and contraction. We recognize two types of data structures,

1. Data structure for adaptive mesh grid points, and
2. Data structure for associated computation data nodes

4.1 Grid Point Data Structure

Maintaining node information is important apart from maintaining node data. Node information might change during the course of execution. (Due to new node creation and re-partition during the computation and load balancing) We designed the data structure for mesh grid points as a linear representation of hierarchical and multi-grid points, generated from a space-filling curve [3,4], which preserves the locality of d-dimensional space mapping to 1-dimensional space,
i.e. N^d - $>N^1$, such that each point in N^d is mapped to a unique point in N^1. Grid point creation, refinement, coarsening, partitioning and dynamic repartitioning operations can be efficiently carried out on the linear representation. The self similar nature of the space filling curve is used to maintain the locality across levels of the grid hierarchy. We used Hilbert space-fillling curve [3] in our experiments. Because of the open architecture the type of the pace-filling curve can be changed as one required, since it is being used as a plug-in to the framework. In addition to the locality preservation, choice of space filling curve is beneficial due to its inexpensive mapping

computational cost. Generated one dimensional representation can be partitioned between processors equally facilitating fast and efficient. The resulting partitions have very good load balancing and edge cut characteristics.

Hilbert-Type Space-Filling Curves

We used the Hilbert's space filling curves [3] for our application. **Figure 4** shows the first 3 stages of Hilbert's space filling curve.

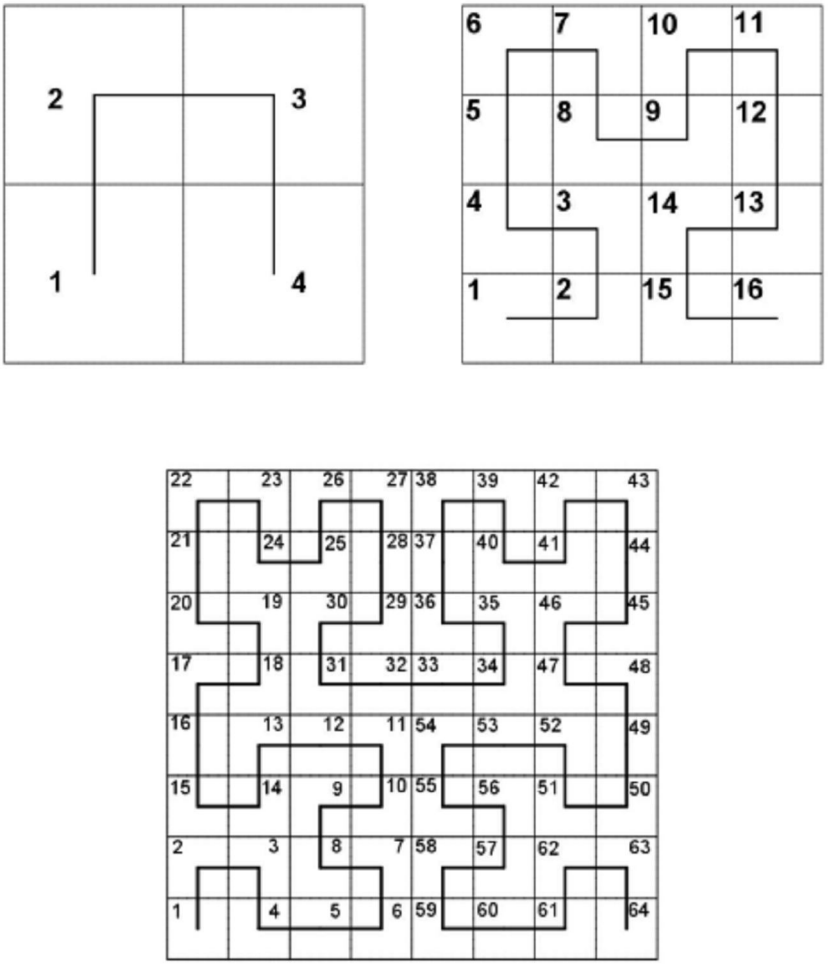


Figure 4: First three stages in the generation of Hilbert's space filling curve [3]

Consider the 2D mesh of 16 nodes in Figure 5. When applied the Hilbert's space filling curve, new order of the one dimensional list would be {0,1,5,4,8,12,13,9,10,14,15,11,7,6,2,3}.

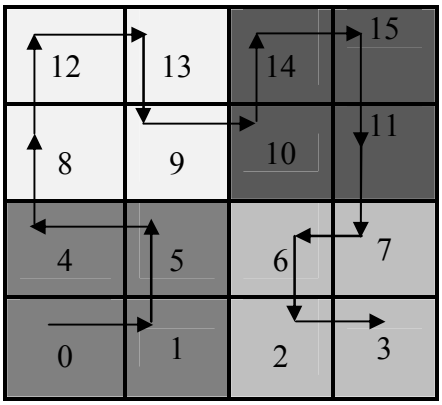


Figure 5: Initial mesh structure of the program graph

When 2D mesh is represented in 1D list, it is easy to partition the grid points among the processors. As seen in the Figure 5 different colors represent the partition received by each processor. This proves that chopping the 1D list will result in consecutive chunks of nodes for each processor. Figure 6 show a refined grid, which has been refined to 3 levels. The space filling curve generated for that graph is

{0,1,{0,1,3,2},4,8,12,13,9,{0,{0,1,3,2},3,2},14,15,11,7,6,2,3}

Each grid point in the mesh will store its coordinates. We name the grid points according to conventional X-Y-Z coordinates. Figure 7 and Figure 8 show the labeling convention of grid point coordinates for 2 dimensional and 3 dimensional meshes respectively. As you can notice from the Figure 7, each set of new refined nodes will consider new X-Y origin, which is bottom-left coordinate of their parent node, to label their coordinates. For a 3 dimensional mesh, the

origin for the new refined grid points will be bottom-left-rear coordinates. This coordinate system is a repetition of original system.

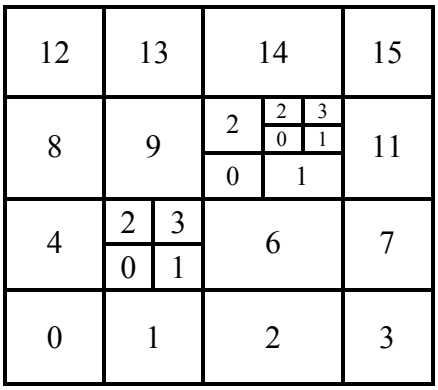


Figure 6: Grid refined to 3 levels

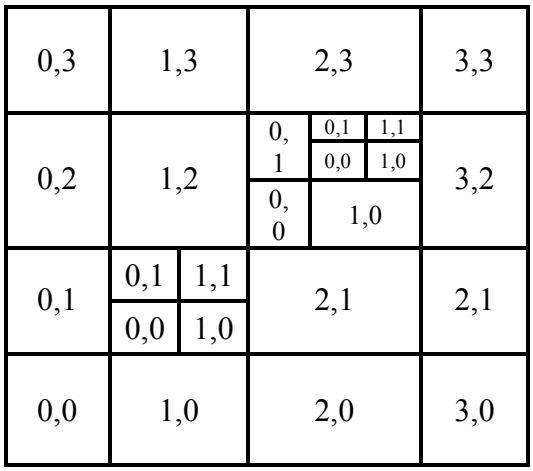


Figure 7: Convention of labeling grid point coordinates (2D)

Each grid point in the mesh is identified by its grid point id. Grid point id represents the location of the grid point as well as its level. Node id for the gray grid point marked in Figure 7 is composed as follows.

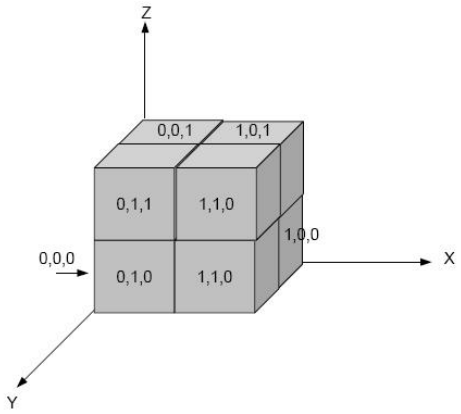


Figure 8: Grid point coordinate naming convention for 3D mesh

In addition to node id, grid data structure keeps the neighboring nodes id list to keep the information about neighbors of the node and shadow node processor information for communication purposes. Two nodes are considered to be neighbors if they share a common edge, if it were 2D list or share a common face, if it were a 3D mesh. Also a node is a shadow node for another node if two nodes are neighbors but belong into two processors. Then first node becomes a shadow node to second and visa versa. If nodes depend on neighbor node information for computation, computation data of shadow nodes are duplicated in appropriate processors. At each communication phase those shadow nodes are updated with current data values for next computation phase. Figure 9 shows the class definition of NodeInfo class, grid point information definition.

```
private:
        int owingProc;
        int level;
        char refine_de_refine; //refine R derefine D none N

public:

        string id;
        set<int> shadowForProcs;
        char peripheral_internal;
        CordinateSystem *system;
        set<string> neighbors;
```

Figure 9: NodeInfo properties

4.2 Computation Node Data Structure

A processor requires fast access to computation data. Thus the natural choice for the data structure holding computation data would be a hash table. Location of the node data is obtained from the node global ID sufficed on modulo hash function. Buckets carry pointers to the node data associate with the global ID. Figure 10 displays how the grid points relate to the computation node data when refining takes place at different levels. Definition of the computation node data is provided by the application user

Entire mesh associated computation data nodes are created and stored in each processor. When node data is required during the computation phase, it will be retrieved using the node id. At the end of communication phase shadow nodes will be sent and incoming shadow nodes are updated in the hash table. In this way, each processor is furnished with required and up-to-date data for its next computation phase.

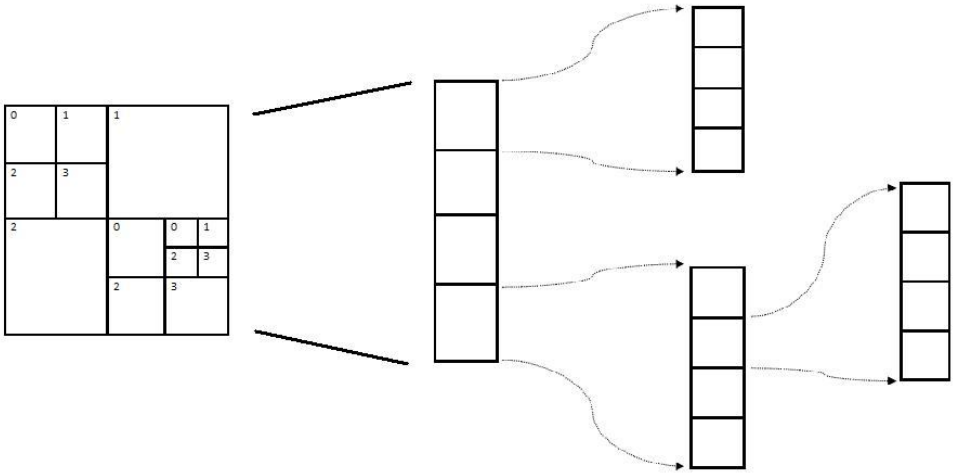


Figure 10: Mapping the adaptive grid to the extended hash table

Each processor set up following lists in local memories.

1. Grid point node list. Processors create and maintain only the grid points belong to own processor. Neighbor information and shadow node information are also maintained with them.
2. Hash map of computation data list. Note that processors keep not only computation node data associated with grid point belong to own processor but computation node data belong to shadow nodes.

5. ALGORITHMS AND IMPLEMENTATION DETAILS

We will elaborate in detail the three phases recognized in the program flow, in this section.

5.1 Initialization Phase

During this phase data structures are created in local memories of the processors to keep graph connectivity, node information and computational node data. Hilbert space-filling curve is called on given mesh structure and generate the one dimensional node list, which is then equally partitioned between the processors to form local node information lists and fill their associated information. Node to processor mapping data will be stored in appropriate data structures. Beside these hash map creation and buffer initialization also take place in this phase.

5.2 Computation and Communication Phase

Each processor performs computation for each of its nodes using its neighbors. After updating the node data, a processor sends the updated shadow nodes information to the appropriate processors. Neighboring processors who receive such information update their data structures to keep up to date for the next computation cycle. Mesh expansion and contraction may occur in this phase as a result of error function. Application developer may define an error function with required criteria for expansion and contraction. According to such criteria application should mark required mesh points to be refined or de-refined. When calculation is done on all grid points framework will call refinement and de-refinement methods on marked nodes. Refer Algorithm 1

for refinement – de-refinment routine. Algorithm 2 shows the node splitting while Algorithm 3 shows the contraction on a grid point. We assume each node will be split into exactly four child nodes. Similarly when contraction occurs, four neighbor nodes in the same level will be reduced to single node. When not all four siblings are marked for de-reinement, coarsening will be ignored on that grid points. At splitting and contraction data node list and node information list will be adjusted accordingly. For an example when new nodes are created as in figure 4 nodes order of the mesh is found out as listed above. In addition to that, neighbors of the previous configuration will be updated with new node information.

```
splitNode()
{
        - Retrieve the DataNode for the expanding nodeId
        - Retrive the location of NodeInfo object of the node
        - Create four new nodes with level = parent level +1.
        - Assign local id and compose the global id from both parent
        id and local id.
        - Fill the neighbor node list and shadow nodes list of new
        nodes.
        - Duplicate parent DataNode data in all four child data nodes.
        - Insert child DataNode to extended hash table
        - Find the space-filling curve order for new child nodes and
        replace parent NodeInfo node from the new child NodeInfo
        nodes list
}
```

Figure 11: Node refining algorithm

Computation function is supplied by the application developer. Framework defined ”computation” function accepts a parameter as a pointer to user supplied computation function. This function

will be called on each grid node belongs to the processor. This allows for a clean and robust decoupling between the framework and the application program code. From the initialization phase, a processor already knows the neighboring processors it needs to communicate the shadow node information to, and the number of such shadow nodes. By the time the computation routine returns, the communication buffers are all set up, and communication can proceed

reduceNodes()

{

- Traverse all “marked for reduction” nodes
- Check if they are in the same level and not level 0
- Check if they are neighbors
- Retrieve DataNode objects of the marked nodes
- Retrive NodeInfo nodes related to each marked nodes
- Create one NodeInfo object for every four neighbor group
- Level of new node = level -1 of new nodes
- Create neighbor list and shadow nodes list for new node.
- Replce each four-group nodes with one DataNode
- Replace four marked nodes in the NodeInfo list with one newly created NodeInfo object.

}

Figure 12: Grid coarsening algorithm

For physical communication of these buffers, new class is defined and committed to an MPI data type (using MPI_Type_commit). All the processors send these buffers at the same time.

MPI_Bsend(), is used for sending these buffers to appropriate processors. MPI_Recv() receives these buffers which are then used to update the locally maintained shadow node information.

```
void computeOverNodes (pointer to application
computation function)
{
  For each node
  {
        invoke application node function( );
        add each marked nodeId to markedList
  }
  For each nodeId in markedList
  {
        call splitNode or reduceNodes functions
   }
}

void communicateShadows()
{
        Create shadow buffers
        for each shadow processor
        {
              MPI_Bsend(shadow nodes);
        }
        for each shadow processor
        {
              MPI_Recv(shadow nodes);
        }
}
```

Figure 13: Computation and communication functions

5.3 Load Balancing Phase

Computational workload changes when new nodes are created or old nodes destroyed at random locations of the mesh. Therefore it is equally important to address the load balancing phase of the framework. Repartitioning should not only balance the work load among the processors but also

keep the edge-cut to a minimum, so as to minimize the inter-processor communication. Figure 14 shows a sample processor grid mapping so that edge cut is kept minimum and the work load is distributed comparably equally. Different shading of the mesh shows, how four different processors have divided the grid points between them.

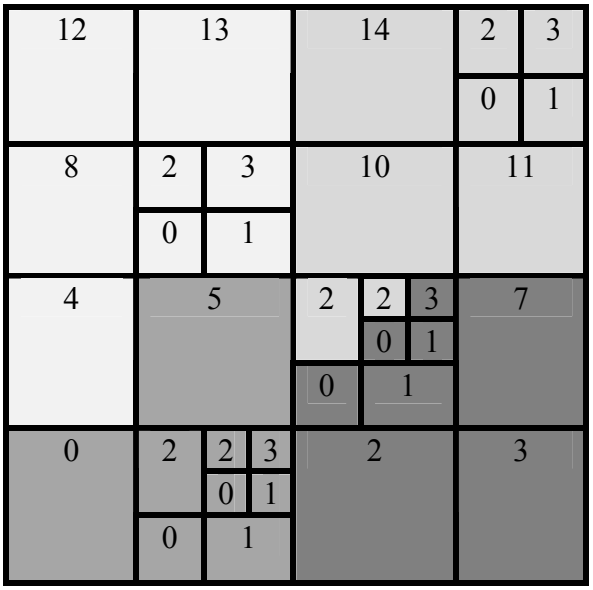


Figure 14: Grid distribution among processors

5.4 Class Description

MPIFramework

Class definition of the framework and implements the MPI calls for send and receive. An instance of the class will be created at each processor, representing the framework. Processor zero will call the Hilbert curve and create the 1 dimensional space filling curve out of n-dimensional mesh. MPIFramework instance of each processor will then create the memory structures from the generated space-filling curve grid point list and find out the own partition in the list. According to

the partitioned list, framework will create the memory structure for NodeInfo objects. And create the hash table for objects dataNode defined by user. According to the program graph information, neighbor lists and shadow node lists are created for each nodeInfo object.

MPIFramework class is responsible for defining computation and communicate functions for the application. In addition to that, MPIFramework defines the send and receive node count, node mapping after receiving from remote processor and load balancing. Figure 15 shows the definition of the MPIFramework class.

```
class MPIFramework
{
private:
        ExtendedHash dataHash;
        vector<NodeInfo> nodeList;
        int procID;

public:
        int numOfProcs;
        MPIFramework(int size);
        ~MPIFramework();
        void computeOverNodes(void (*simulator_ptr)(NodeInfo* node ));
        void communicateShadows(int* sendCountArray);
        void init(int size);
        int* createShadowCountToSend(int* sendCountArray);
        void MPIFramework::loadBalance();
};
```

Figure 15: MPIFramework class definition

NodeInfo

This class defines a grid point. Each grid point is recognized with a unique id, each grid point lies on a certain grid level. When the grid get finer new grid points will be created and old grid point will be replaced. When further fine grids are required, grid points at those locations will be further splitted into smaller grids. On the other way hand grid points will be reduced to fewer grid points

when it is required to be coarser. This process is continued in hierarchical order. Therefore each grid point is defined to be in a certain level and this property is stored in NodeInfo class. Other than those properties NodeInfo class keeps the neighbor list and shadow nodes list of each nodeInfo object.

```
class NodeInfo
{
private:
	int owingProc;
	int level;
	char refine_de_refine; //refine R derefine D none N

public:

	string id;
	set<int> shadowForProcs;
	char peripheral_internal;
	CordinateSystem *system;
	set<string> neighbors;
	NodeInfo(string globalId,int procID);
	~NodeInfo();
	void setInitialNeighbors(int x, int y);
	void setInitialNeighbors(int x, int y, int z);
	void setShadowNodes(map<string,int> nodsProcMap);
	int getLevel();
	void setRefine();
	void setDerefine();
	void resetRefineDerefine();
	char isRefinedDerefined();
	void setPeripheralOrInternal(char nodeType);
	void setLevel(int parentLevel);
	void splitNode(int meshLength, vector<NodeInfo>& nodeList,map<string,int>
	    nodesProcMap, ExtendedHash& hashMap);
};
```

Figure 16: NodeInfo class definition

NodeInfo class defines the splitting and reducing methods required for mesh refinement. According to the error function, application developer can set refine or derefine property of the

nodeInfo object so that framework can later call split or reduce function on the object to obtain finer or coarser grids. Figure 16 shows the NodeInfo class definition.

ExtendedHash

This class is defined to create a hash table to keep data objects as hash entries. Key to the hash entry is the nodeId and value of a hash entry is a pointer to the NodeData object. ExtendedHash class facilitates the dynamic insert; delete which are beneficial in mesh refinements. Essentially hash entry is important since its nature of fast access to the data through hash keys.

```
class ExtendedHash
{
private:
        int depth;
        map<string, DataNode*> hashMap;
        int getKey(char* key);

public:
        ExtendedHash();
        ~ExtendedHash();
        DataNode * getElement(string key);
        void deleteElement(string key);
        void insertElement(string key,DataNode* dataNode);
};
```

Figure 17: ExtendedHash class definition

NodeData

NodeData class defines the application grid point class which keeps the data values involved in calculations.

```
class DataNode
{
private:
        int id;
        int data;

public:
        DataNode();
        ~DataNode();
        DataNode* createClone(DataNode& dataNode,  DataNode &
cloneNode);
};
```

Figure 18: DataNode class definition

BuffNode

This class is defined for the communication purposes. Fields which should be transferred to the other processor will be defined in this class to make it light in weight. Application developer should change this class according to its application grid point class.

```
class BuffNode
{
public:
        char nodeId[NAME_LENGTH];
        DataNode dataNode;

        BuffNode();
        ~BuffNode();
};
```

Figure 19: BuffNode DataNode class definition

Graph

Graph class handles the mesh structure. It reads the output file generated from the Hilbert space-filling curve to create data structure and load to the memory.

```
class Graph
{
public:
        Graph();
        ~Graph();
        vector<string> getGraph(const char* fileName, int size);

        void nodeProcMap(map<string,int> & procMapping,vector<string> nodeList, int
noOfprocs);

};
```

Figure 20: Graph class definition

In addition to that, Graph class is responsible for mapping the grid points and the processor id to which the grid points belong. Figure 20 shows the Graph class definition.

FileIO

FileIO class is responsible for reading the output file generated from Hilbert space-filling curve call.

```
class FileIO
{
public:
      FileIO();
      ~FileIO();
      vector<string> getOutput(const char* fileName);
};
```

Figure 21: FileIO class definition

Point

Point is the base class for 2D and 3D point classes which define the 2 dimensional and three dimensional points respectively.

```
class Point {

public :
        int x;
        Point(){ };
        ~Point(){};
};
```

Figure 22: Point class definition

Figure 23 and Figure 24 show the two dimensional and three dimensional points' definitions respectively. One of these class definitions will be called to create point objects depending on the dimension of application porting.

Point2D

```
class Point2D : public Point {

public:
        int y;
        ~Point2D(){};
        Point2D(int ix, int iy)
       {
                x =ix;
                y =iy;
        };
};
```

Figure 23: 2D point class definition

```
class Point3D : public Point {
public:
	int y;
	int z;
	~Point3D()
	{
	};
	Point3D(int ix, int iy, int iz)
	{
		x =ix;
		y = iy;
		z = iz;
	};
};
```

Figure 24: 3D point class definition

CordinateSystem

This class is used to store the coordinates of each grid point. Two dimensional and three dimensional coordinate systems are derived from this class.

```
class CordinateSystem {

public:

	virtual void setCordinates(Point2D initPoint, int level, int x, int y)=0;
	void setCordinates(Point3D initPoint, int level, int x, int y, int z){};
	bool isNeighbor(vector<Point2D> cordinates){return false;}
	bool isNeighbor(vector<Point3D> cordinates){return false;}
};
```

Figure 25: Coordinate system class definition

System2D

Defines the coordinate system for two dimensional meshes.

```
class System2D: public CordinateSystem
{
public:
	vector<Point2D> cordinates;
	System2D();
	~System2D();
	void setCordinates(Point2D initPoint, int level, int x, int y);

	bool isNeighbor(vector<Point2D> cordinates);
};
```

Figure 26: Coordinate system for 2D meshes

System3D

Defines the coordinate system for three dimensional meshes.

```
class System3D: public CordinateSystem
{
public:
	vector<Point3D> cordinates;
	System3D();
	~System3D();
	void setCordinates(Point3D initPoint, int level, int x, int y, int z);
	void setCordinates(Point2D initPoint, int level, int x, int y){};
	bool isNeighbor(vector<Point3D> cordinates);
};
```

Figure 27: Coordinate system for 3D meshes

Please refer Appendix A for the complete class diagram of the system and Appendix B for the source code.

5.5 Overheads

When designing a generic platform, an overhead will be introduced definitely. One of the key issue is that how do we invoke the application specific functions on adaptive mesh and enable user to call error function on them. Application user is required to pass a pointer to the computation

function which accepts following parameters: pointer to the grid point, on which we call the computation function, hash table containing computation data. Since grid point is passed to the function, application user can access the neighbors and shadow nodes list of the grid point. Also user can access the hash map using global node id of grid point, to get the associated computational data. Computation function should contain an error function to decide the refining and coarsening criteria. When the criteria is met user can mark the grid points to be refined or de-refined. Buffer creation and send and receiving of grid point information and computation node information is again an overhead. Also, an overhead will be introduced at neighbor information and shadow information filling whenever a grid-restructuring occurs. During the calculation, load balancing checking is done. This could add few overhead processor cycles to the application flow. Finally, during the load balancing phase, we call few MPI_Send and MPI_Recv routines to collect required information and calculate the load distribution. We tested have tested the overhead introduced from the framework and have received the results to be at an acceptable level. Refer Section 6 for experimental results.

6. EXPERIMENTAL RESULTS

The experiments were conducted on Silicon Graphics Origin- 2000 computer with 24 CPUs with hypercube cc-NUMA architecture. We used two applications, one for 2D mesh for solving the Partial Differential Equation with Euler and Heun, developed by Kurt Bingham [21] and one for 3D mesh for acoustic waveforms developed with Clawpack [23] for our experiments. First we consider 2D mesh application. Original application is a VB.net program and later converted into C++ language. The original application does not support the adaptive mesh refinement. Our framework has successfully added the joy of experiencing adaptive mesh refinement on application.

6.1 Case Study: Polytrope Differential Equation Solver

A polytrope is a model for gaseous mass whose thermodynamics can be approximated by

$P = K\rho^{((n+1)/n)}$

Where P is pressure, ρ is density and K is a constant. The constant n is known as the polytropic index. Gas following such an equation of state produces a polytropic solution to the Lane-Emden equation.

The Lane-Emden Equation in familiar differential equation notations,

$$y'' = -y^n - 2\frac{y'}{x}$$

The equation we are trying to solve, when written as an Euler's Equation for hydrostatic equilibrium of a star is,

$$\frac{dP}{dr} = \frac{-GM(r)}{r^2}\rho(r)$$

Where ρ(r) density, M(r) mass and P(r) pressure as functions of r which is the distance from the center of the star.

To get the initial values for y and y' Taylor expansions is used.

$$y(e) = 1 - \frac{e^2}{6} + \frac{n}{120}e^4$$

$$y'(e) = -\frac{e}{3} + \frac{n}{30}e^3$$

Seed value (e) is very small.

The solutions calculated from the above Tailor expansions are then considered to be scalar known quantities.

$$y(e) = A$$
$$y'(e) = B$$

Equations are solved for the position where F becomes zero, i.e. the pressure/density crosses the x-axis. This point is a point on surface. The application consists of a second order partial differential equation solver which is incorporated into a polytrope class. The solver can use either Euler or Heun [26] methods.

Table 1 shows the time taken for 300 iterations of mesh sizes 8X8, 16X16, 64X64, 128X128 and 256X256 on 2 processors without mesh re-gridding (without adaptive performance). The average time is recorded from all four processors. Table 2 shows the execution time taken for 8X8, 16X16, 64X64, 128X128 and 256X256 on two processors with re-grinding.

Table 1: Execution time (in seconds) on 2 processors without adaptive refinement

Mesh Size	8X8	16X16	64X64	128X128	256X256
Execution Time	0.4064	1.6433	23.8125	95.165	381.587

Table 2: Execution time (in seconds) on 2 processors with adaptive refinement

Mesh Size	8X8	16X16	64X64	128X128	256X256
Execution Time	1.787	22.88	97.09	386.81	1532.2

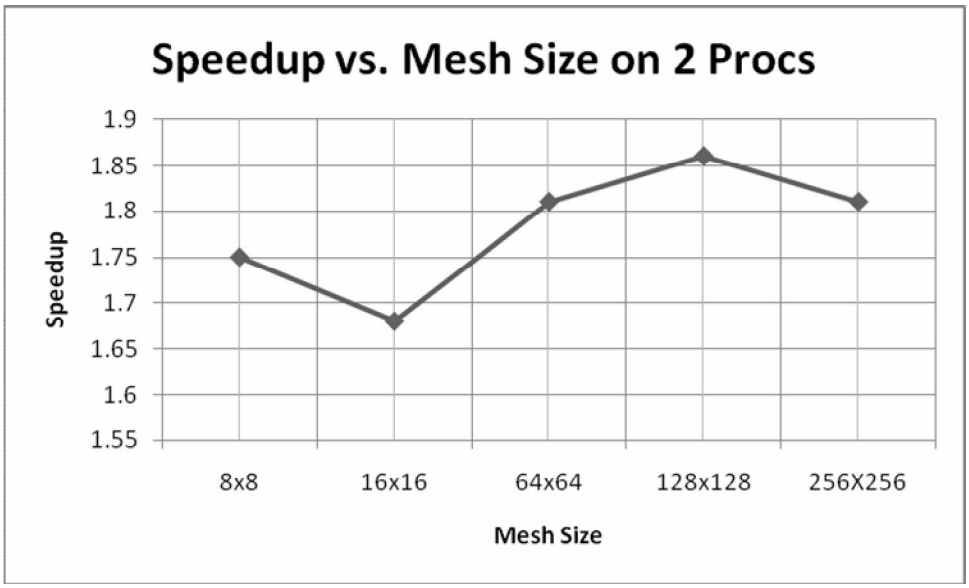


Figure 28: Speedup vs. mesh size on 2 processors without adaptive refinement

Table 3 and 4 shows the timing recorded for 300 iteration of Euler computation on mesh sizes 8X8, 16X16, 64X64, 128X128 and 256X256 on four processors with and without adaptive behavior respectively.

Table 3: Execution time (in seconds) on 4 processors without adaptive refinement

Mesh Size	8X8	16X16	64X64	128X128	256X256
Execution Time	0.2243	0.8663	12.359	49.4975	192.729

Table 4: Execution time (in seconds) on 4 processors with adaptive refinement

Mesh Size	8X8	16X16	64X64	128X128	256X256
Execution Time	0.932	13.12	50.92	192.98	756.32

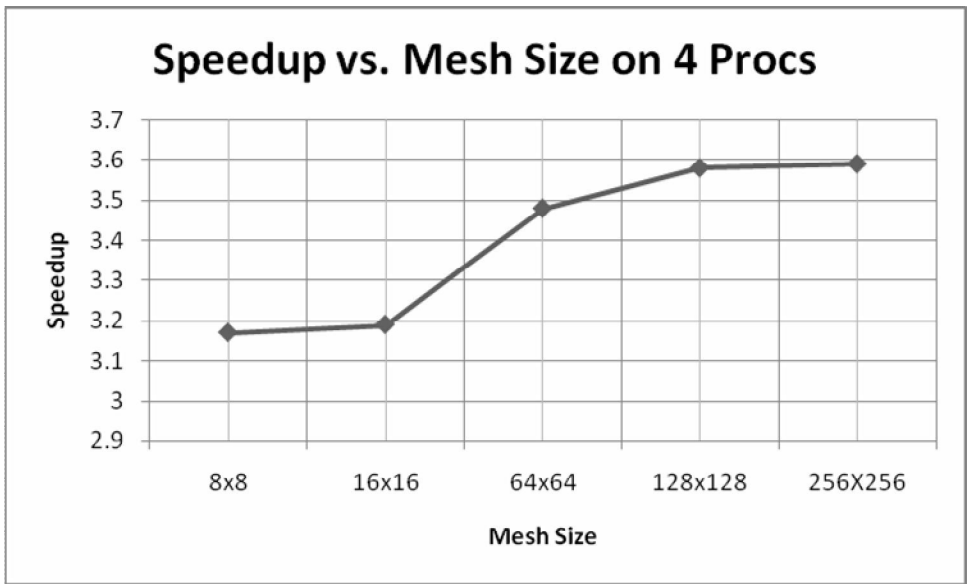


Figure 29: Speedup vs. mesh size on 4 processors without adaptive refinement

Figure 23 shows the speedup achieved at each mesh size. We observed a decrease of speed up increase as mesh size grows. Tables 5 and 6 give the execution time on 8 processors with similar conditions given as for the above cases.

Table 5: Execution time (in seconds) on 8 processors without adaptive refinement

Mesh Size	8X8	16X16	64X64	128X128	256X256
Execution Time	0.1386	0.4315	6.6648	26.6942	105.631

Table 6: Execution time (in seconds) on 8 processors with adaptive refinement

Mesh Size	8X8	16X16	64X64	128X128	256X256
Execution Time	0.4587	6.8753	27.344	107.09	437.87

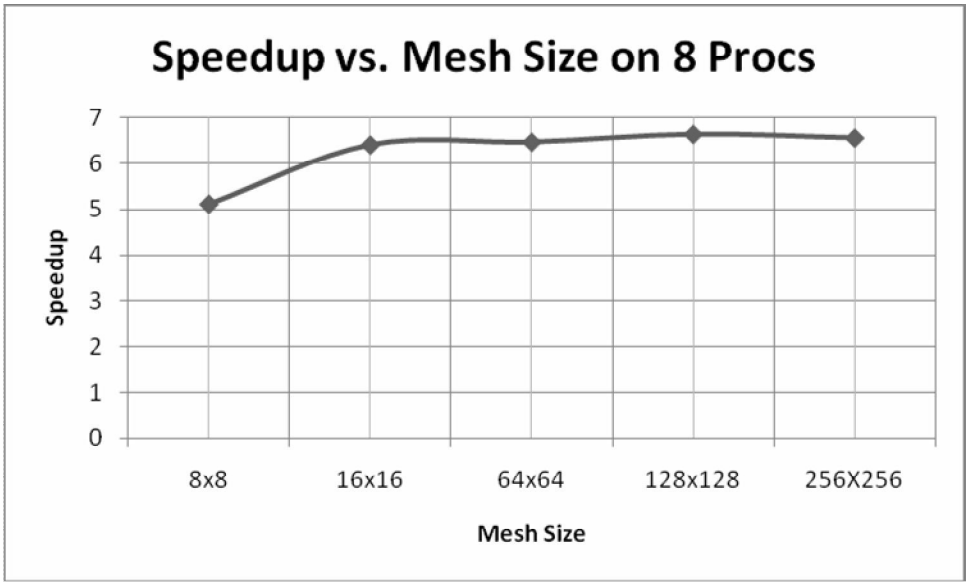


Figure 30: Speedup vs. mesh size in on 8 processors without adaptive refinement

Figure 24 shows the speedup plot obtained. We observed about 6 to 7 speedup when the size of the mesh size increases when run on 8 processors. Further we achieved up to 11 times speedup when tested on 16 processors. Figure 25 shows the plot drawn for speedup vs. mesh size.

Table 7: Execution time (in seconds) on 16 processors with adaptive refinement

Mesh Size	8X8	16X16	64X64	128X128	256X256
Execution Time (s)	0.3987	3.2671	15.875	57.34	222.84

Table 8: Execution time (in seconds) on 16 processors without adaptive refinement

Mesh Size	8X8	16X16	64X64	128X128	256X256
Execution Time (s)	0.0911	0.2757	3.7897	15.3324	56.3959

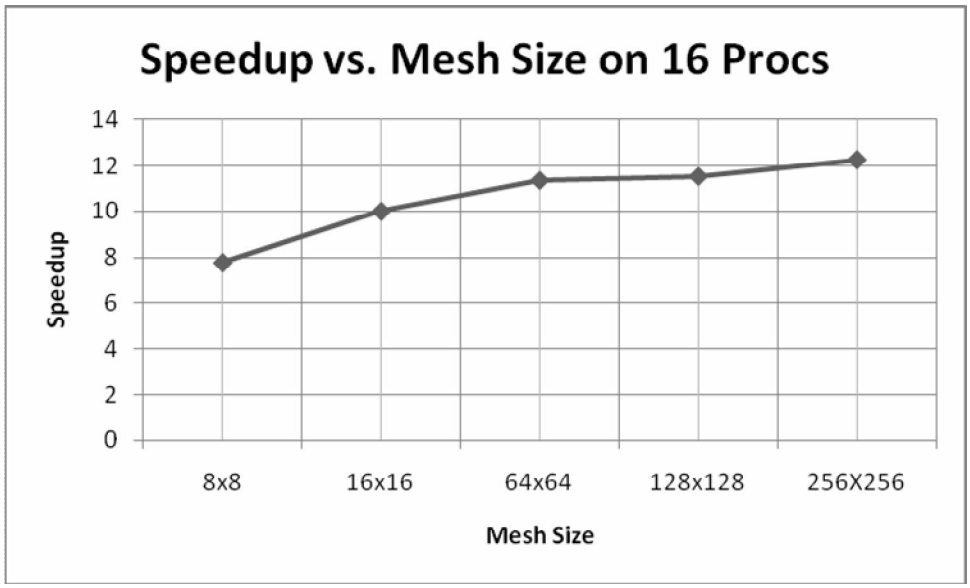


Figure 31: Speedup vs. mesh size on 16 processors without adaptive refinement

Figure 26 shows the timing speeds up of achieved with different number of processors. We used 64X64 and 128X128 meshes with 300 iterations on Euler method.

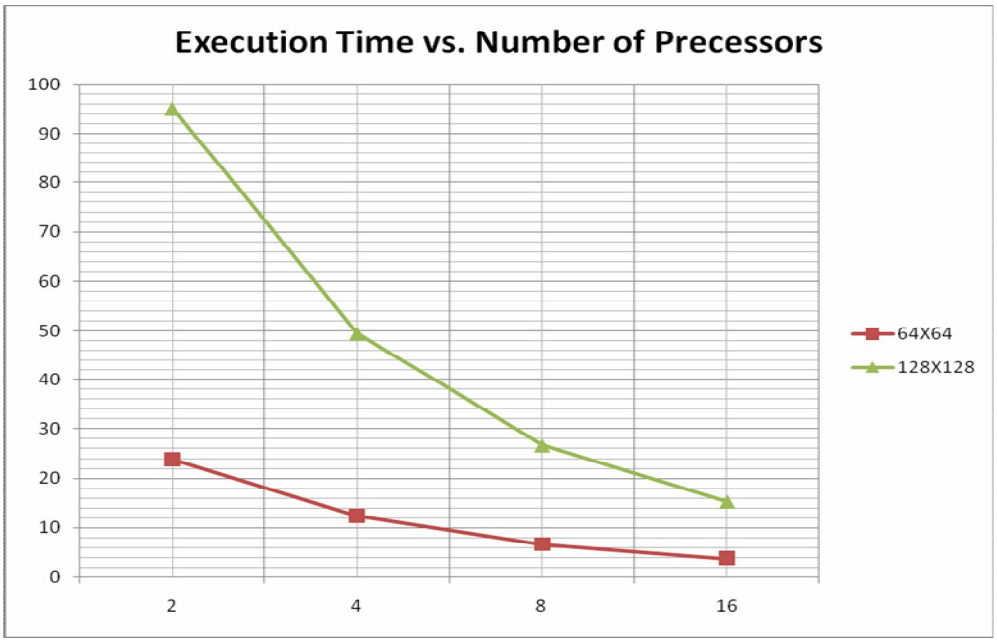


Figure 32: Execution time vs. number of processors without adaptive refinement

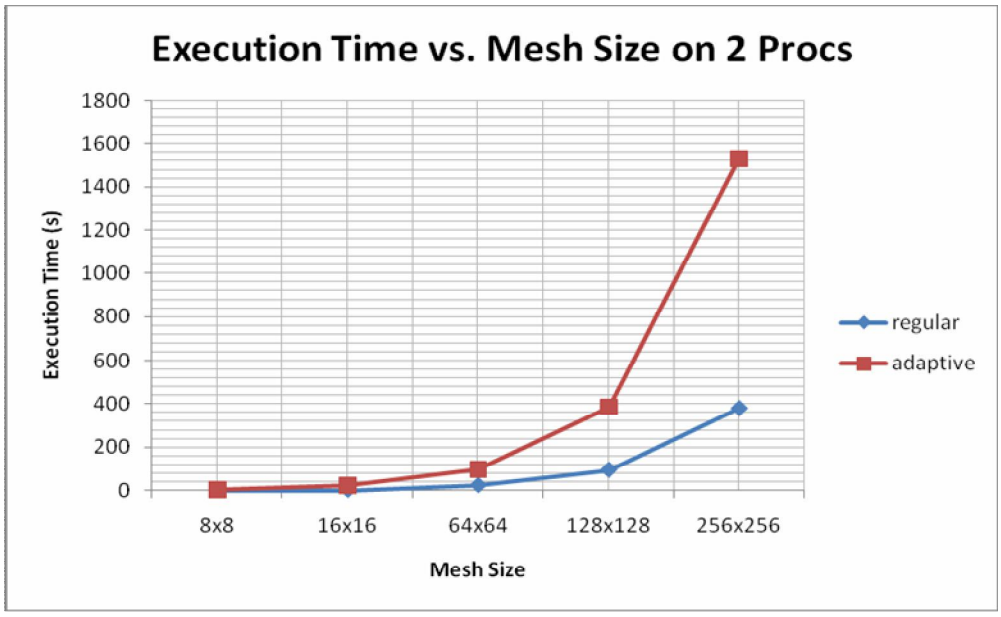


Figure 33: Execution time vs. mesh size on 2 processors with regular/adaptive refinement

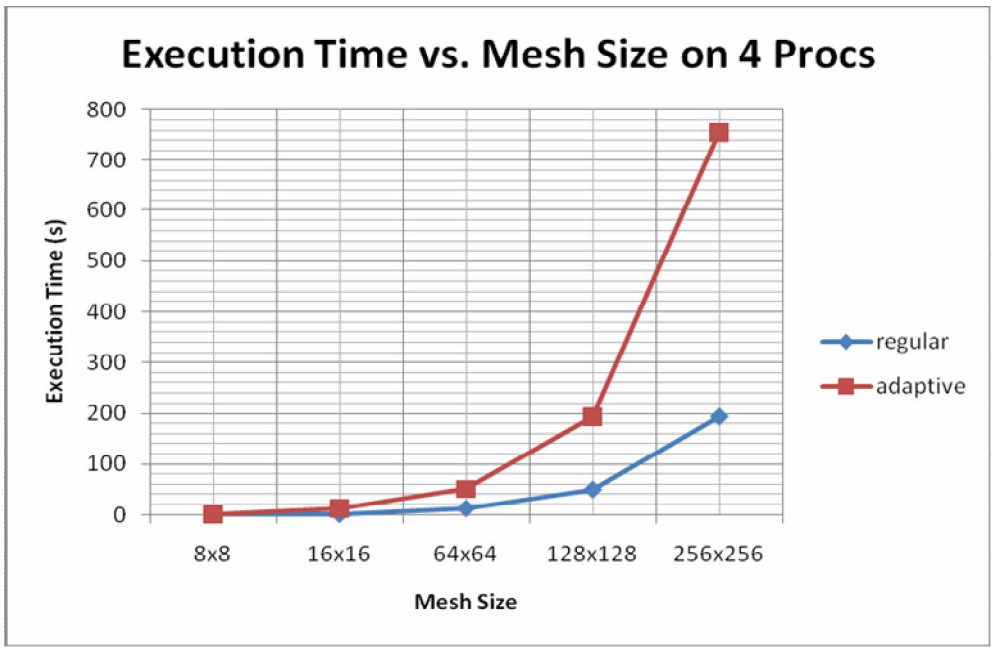


Figure 34: Execution time vs. mesh size on 4 processors with regular/adaptive refinement

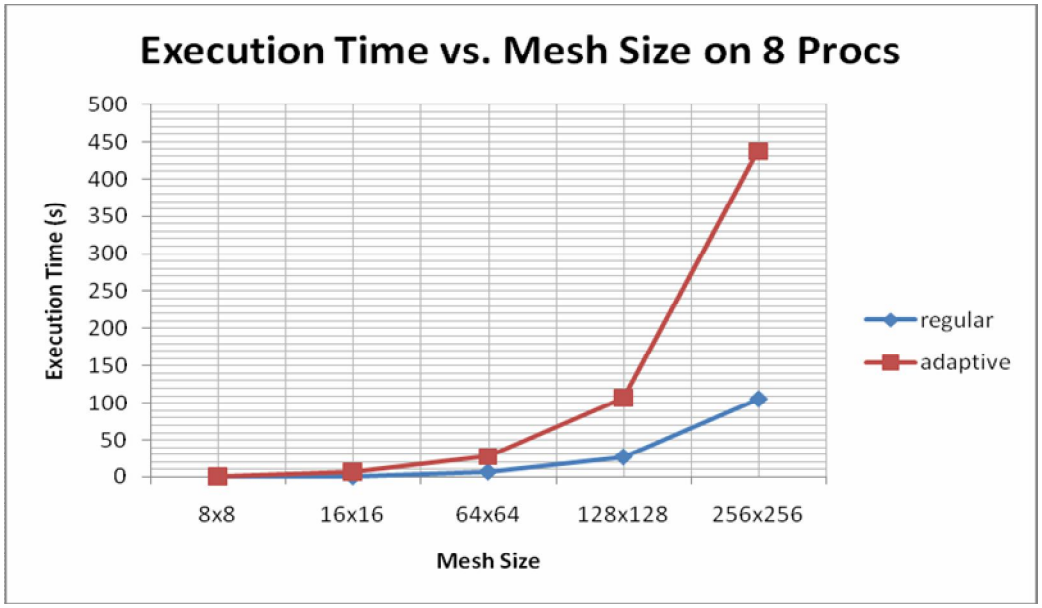


Figure 35: Execution time vs. mesh size on 8 processors with regular/adaptive refinement

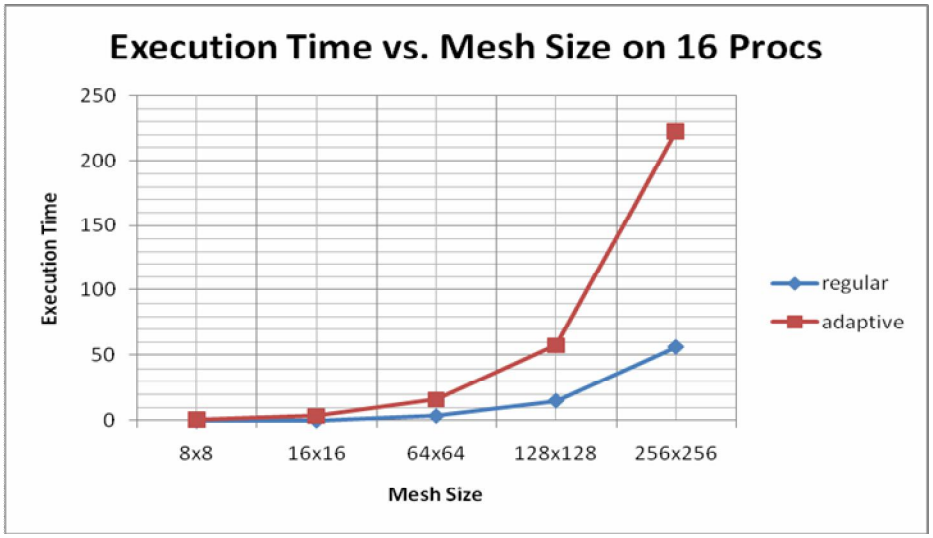


Figure 36: Execution time vs. mesh size on 16 processors with regular/adaptive refinement

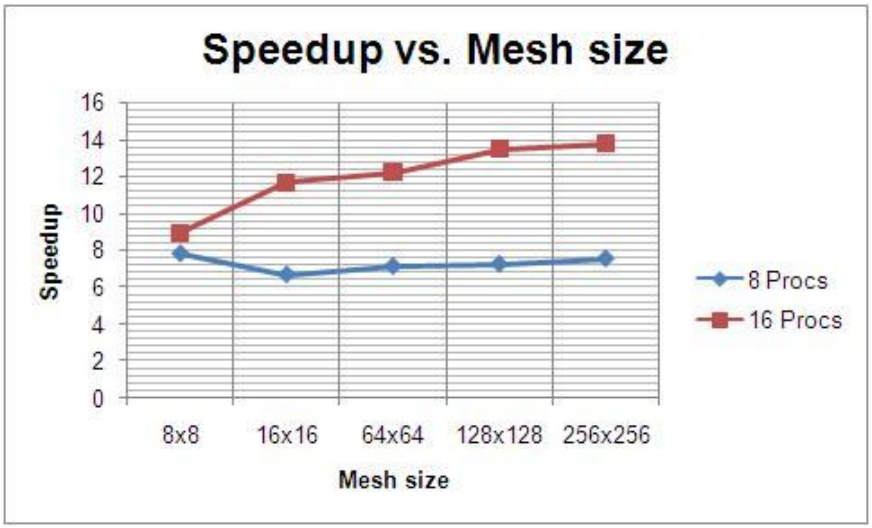


Figure 37: Relative speed up vs. mesh size when adaptive behavior is enabled

We plot in Figure 37, speed up of application with adaptive mesh behavior on 8X8, 16X16, 64X64, 128X128 and 256X256 meshes on 8 and 16 processors. This is a approximated behavior we can expect. Because the original application does not support the adaptive mesh behavior, we calculated the sped up with framework ported application run on single processor.

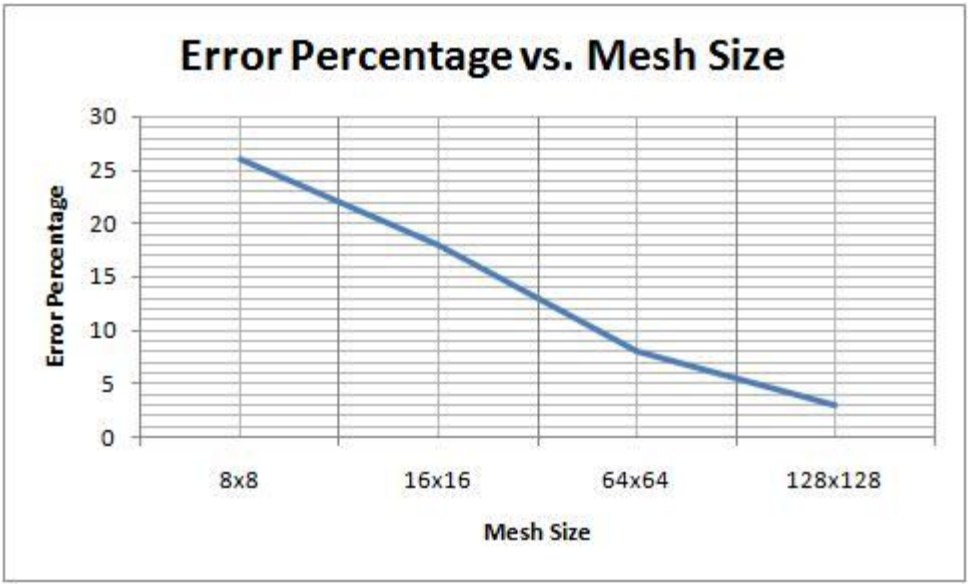


Figure 38: Error rate of the application on different mesh sizes

Application calculates various properties such as pressure, density etc. of very large collection of points in space inside the polytrope. The application can be utilized to find properties and behavior of strange mass/radius combinations.

We experiment the adaptive mesh of size 16 X16 on different processor, and at some instances recorded load on each processor. Average of different instances resulted in Table 9 for normalized load metric.

Table 9: Normalized load metric and number of processors

Number of Processors	1	2	4	8
Normalized Load Metric	1	0.85	1.3	1.6

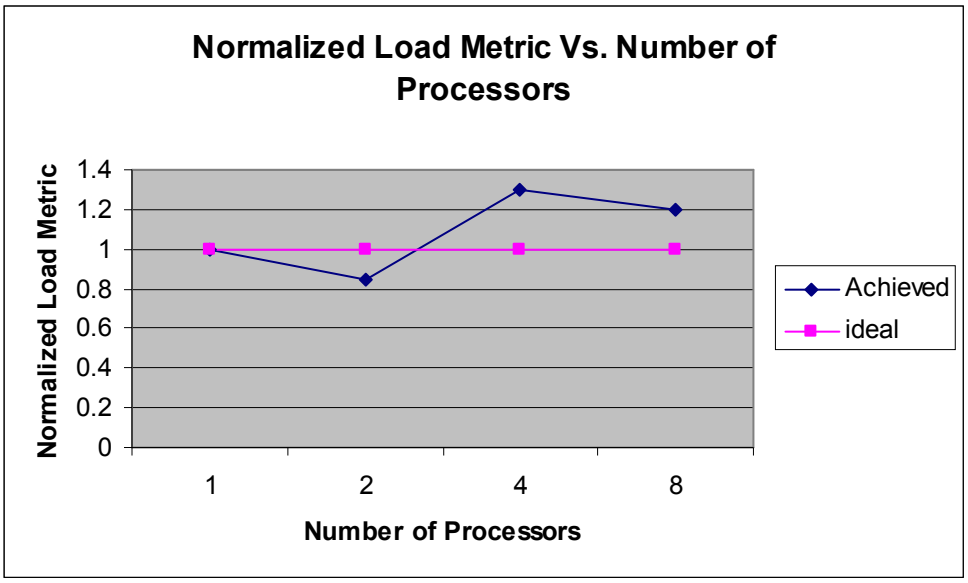


Figure 39: Normalized load metric on different processors

We explored the overhead introduced by the framework. We can recognize four main parts of the framework in terms of functionalities: initialization, computation, communication and load balancing. Figure 36 shows the percentage of time taken from each part when 128X128 mesh is employed on 4 processors. As Figure 40 dipicts considerable amount of time is consumed for computation.

Table 10:Time taken for different execution units

Execution Unit Name	Execution Time (s)
Initialization	7.66
Computation	82.06
Communication	5.44
Load balancing	5.45

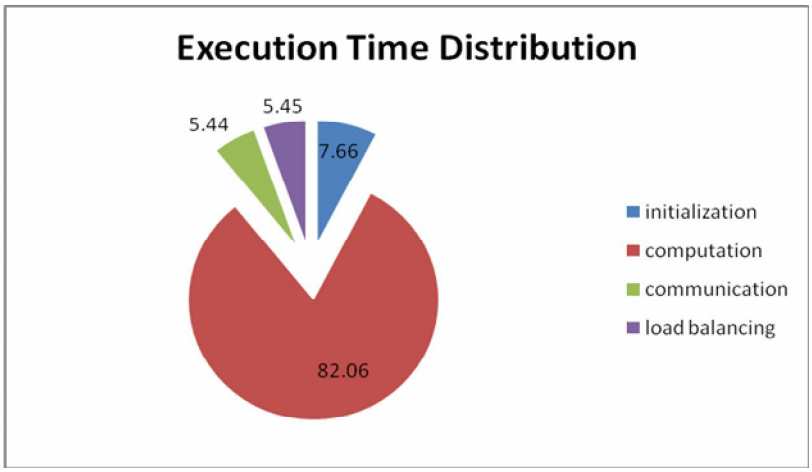


Figure 40: Percentage time taken for different execution unit on 128X128 mesh

We compared the performance of framework when above application is ported, with PETSc [30] when the same application is ported. Figure 41 shows the execution time achieved.

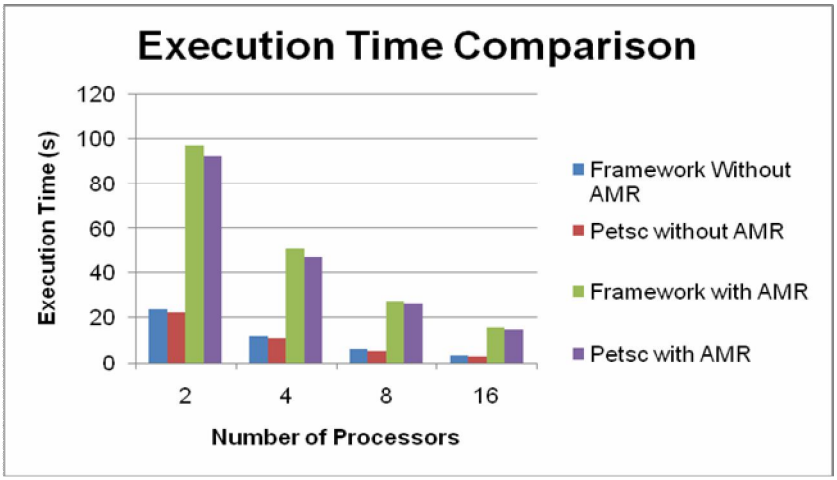


Figure 41: 64x64 mesh execution time comparison with PETSc and framework

6.2 Case Study: Wave Propagation in Media

This is a three dimensional mesh implementation by CLAW [37] to analyze the wave propagation behavior in media.

We conducted experiments porting their application on 32x32x32 mesh and 64x64x64 mesh. Experiment setup was as follows. Four equations were considered in hyperbolic system and two waves are assumed in each Reimann solutions. To run the experiments we used a system with Silicon Graphics Origin- 2000 computer with 24 CPUs as described at the beginning of the section 6.

Figure 42 and Figure 43 shows the speedup achieved on respective mesh sizes when AMR is not enabled.

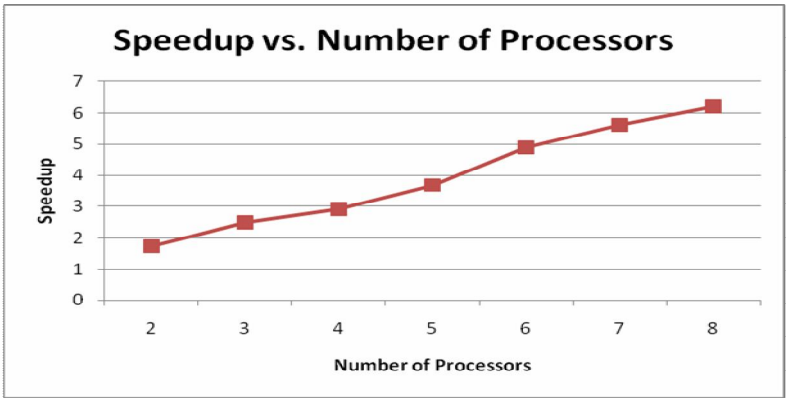


Figure 42: Speedup vs. processors on 32x32x32 mesh

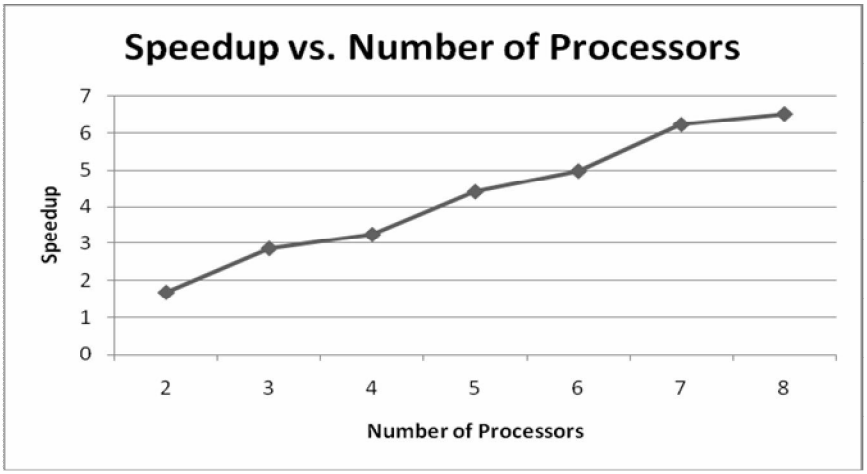


Figure 43: Speedup vs. processors on 64x64x64 mesh

Following experiments were carried out with AMR behavior enabled on both implementations: CLAWPACK and our framework. Figure 44 and Figure 45 shows the achieved speed up porting to the framework.

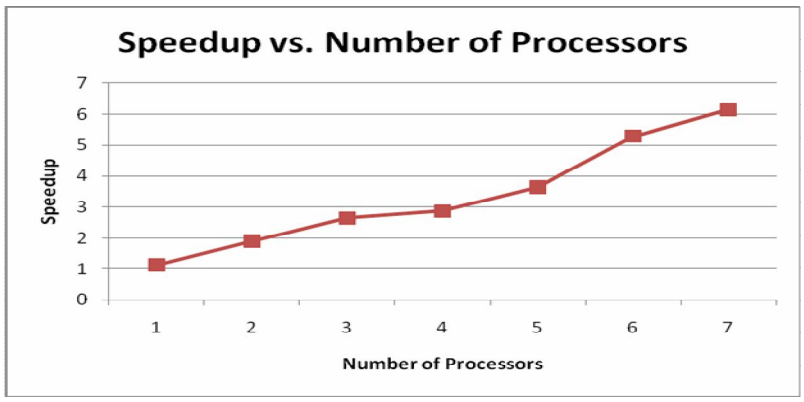


Figure 44: Speedup vs. processors on 32x32x32 mesh with adaptive refinement

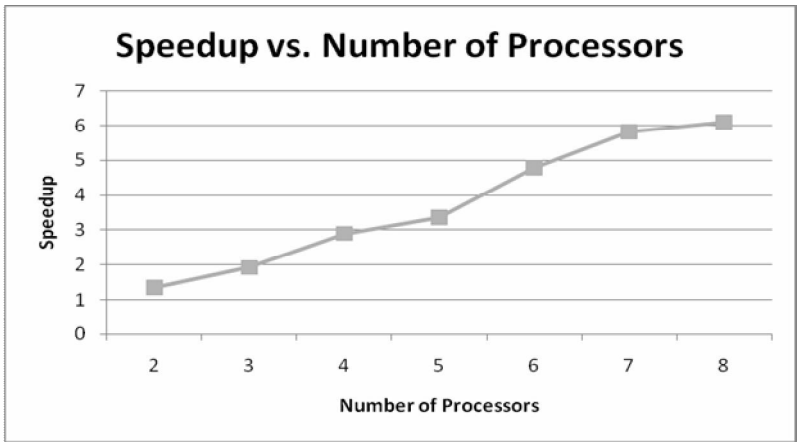


Figure 45: Speedup vs. processors on 64x64x64 mesh with adaptive refinement

We have run a toy application on framework for different mesh sizes and plot the behaviour of the framework in Figure 46 . We ran the framework on four processors. Toy application does no

computation but initialization of grid structure, initialization of computation data, filling shadow nodes list and setting up communication bufferes

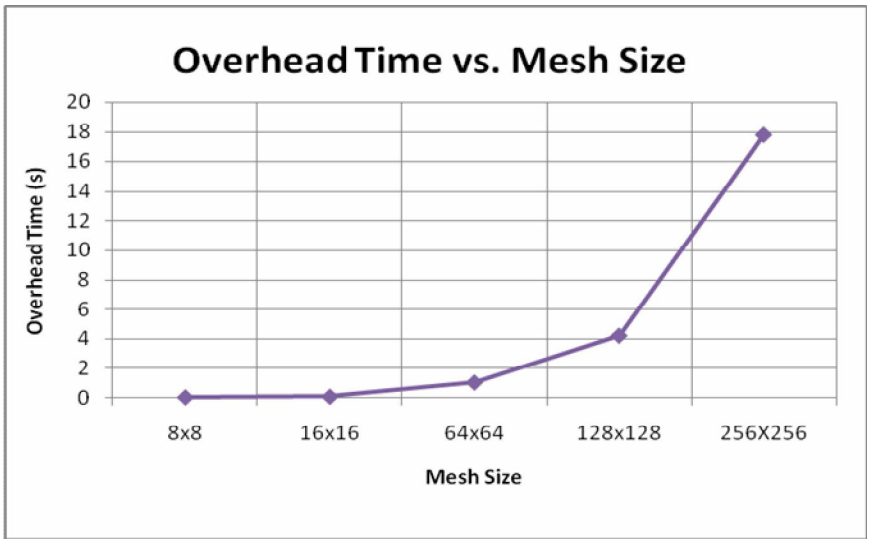


Figure 46: Overhead introduced from the framework

7. CONCLUSION AND FUTURE WORK

The methodology presented in this thesis provides a strategy for parallelization of adaptive mesh structured applications. We have presented efficient way of keeping data related to the mesh application, dynamic algorithms for grid expansion, contraction and load balancing. It is relatively easy transition to an application developer from sequential application to MPI parallel application, being just few inputs required by the framework. We demonstrated the performance of our framework with 2 dimensional and 3 dimensional applications: an application solving Partial Differential Equation with Euler and an application for acoustic waveforms. Our load balancing algorithm also demonstrates good performance in balancing the load. We believe with very little changes, we can improve the performance of our framework further. Future work may include making the framework enable to plug-in third party load balancing algorithms, and mapping the processor architecture to mesh architecture. Also we require an overlap of computation and communication phase so that we can further reduce the overhead. At the same time we are exploring the possibility of integrating our previous work iC2mpi[18] so that the framework should be able to port graph structured application to the framework opening huge domain of programs to get the benefit of the platform.

REFERENCES

[1] MPI: A Message-Passing Interface Standard, Version 2.1, Message Passing Interface Forum, June 23, 2008.

[2] Hans Sagan, *Space Filling Curves*, Springer_Verlag , New York, 1994

[3] Nicholas J. Rose, *Hilbert-Type Space-Filling Curves,*

[4] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger and H. Raymond Strong, *Extendible Hashing-A Fast Access Method for Dynamic Files*, ACM Transaction on Database Systems, Vol. 4, No. 3, September 1979

[5] J¨orn Behrens and Jens Zimmermann, *Parallelizing an Unstructured Grid Generator with a Space-Filling Curve Approach*

[6] R. Henderson, D. Meiron, M. Parashar and R, Samtaney, "Parallel Computing in Computational Fluid Dynamics". In J. Dongarra et al., editors, *"Sourcebook of Parallel Computing",* Chapter 5, Morgan Kaufmann, 2003.

[7] I.M. Smith and D.V. Griffiths : *Programming the finite element method,* (John Wiley & Sons, 2004, 4th edn.)

[8] Michael J. Quinn, *Parallel Programming in C with MPI and OpenMP,* McGraw Hill, 2004.

[9] Marsha J. Berger and Joseph E. Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations,* Stanford University Stanford, CA, USA , Technical Report: NA-M-83-02, 1983.

[10] Jose G. Castanos and John E. Savage, *"The Dynamic Adaptation of Parallel Mesh-Based Computation",* Technical Report: CS-96-31, 1996.

[11] http://seesar.lbl.gov/ccse/Software/index.html

[12] A. Osman , H. Ammar, *Dynamic Load Balancing Strategies for Parallel Computers*, International Symposium on Parallel and Distributed Computing (ISPDC) 2002.

[13] Xiang Yang Li and Shang-Hua Teng, *Dynamic Load Balancing for Parallel Adaptive Mesh Refinement*, 5th International Symposium on Solving Irregularly Structured Problems in Parallel 1998.

[14] Bishwaroop Ganguly and Andrew Chien, *High-Level Parallel Programming of An Adaptive Mesh Application Using the Illinois Concert System*, Department of Computer Science, University of Illinois.

[15] Youhui Zhang, Dan Pei, Dongsheng Wang and Weimin Zheng, *A Task Migration Mechanism for MPI Applications*, Department of Computer Science, Tsinghua University, Beijing 100084, China.

[16] William George, *Dynamic Load-Balancing for Data-Parallel MPI Programs*, Message Passing Interface Developer's and User's Conference (MPIDC'99) 1999.

[17] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger, *Adaptive Load Balancing for MPI Programs*, Center for Simulation of Advanced Rockets, University of Illinois at Urbana-Champaign.

[18] Botadra, H., Cheng, Q., Prasad, S.K., Aubanel, E., Bhavsar, V., *iC2mpi: A Platform for Parallel Execution of Graph-Structured Iterative Computations*, Parallel and Distributed Processing Symposium, 2007. IPDPS 2007.

[19] Manish Parashar and James C. Browne, *Distributed Dynamic Data-Structures for Parallel Adaptive Mesh-Refinement,* Proceedings of the International Conference for High Performance Computing 1995.

[20] Charles A. Rendleman, Vincent E. Beckner, Mike Lijewski, William Crutchfield and John B. Bell, *Parallelization of Structured, Hierarchical Adaptive Mesh Refinement Algorithms*, Lawrence Berkeley National Laboratory.

[21] R. Cappuccio , G. Cattaneo , G. Erbacci , U. Jocher, A parallel implementation of a cellular automata based model for coffee percolation, *Parallel Computing,* v.27 n.5, p.685-717, April 2001.

[22] S. Roberts, S. Kalyanasundaram, M. Cardew-Hall and W. Clarke *A Key Based Parallel Adaptive Refinement Technique for Finite Element Methods*.

[23] A. S. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. L. Welcome. *A conservative adaptive projection method for the variable density incompressible Navier-Stokes equations*. J. Comput. Phys., 142:1–46, May 1998.

[24] D. E. Stevens, A. S. Almgren, and J. B. Bell. *Adaptive simulations of trade cumulus convection* submitted for publication, 1998.

[25] R. B. Pember, L. H. Howell, J. B. Bell, P. Colella, W. Y. Crutchfield, W. A. Fiveland, and J. P. Jessee, *An adaptive projection method for unsteady low-Mach number combustion,* Comb. Sci. Tech., 140:123–168, 1998.

[26] http://spiff.rit.edu/classes/phys317/lectures/heun/heun.html

[27] Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci and Daniel A. Reed, *Autopilot: Adaptive Control of Distributed Applications*, High Performance Distributed Computing, 1998. The Seventh International Symposium.

[28] J. Pruyne and M. Livny. *Parallel Processing on Dynamic Resources with CARMI*.

[29] The Charm++ Programming Language Manual, Parallel Programming Laboratory University of Illinois at Urbana-Champaign.

[30] Portable, Extensible Toolkit for Scientific Computation, PETSc, www.mcs.anl.gov/petsc/petsc-as

[31] Milind Bhandarkar and L. V. Kale. *A Parallel Framework for Explicit FEM*, In proceedings of the International Conference on High Performance Computing, Bangalore, India, December 2000.

[32] http://libmesh.sourceforge.net

[33] Sirinivas Aluru, Faith E. Sevilgen, *Parallel Domain Decomposition and Load Balancing Using Space-Filling Curves.*

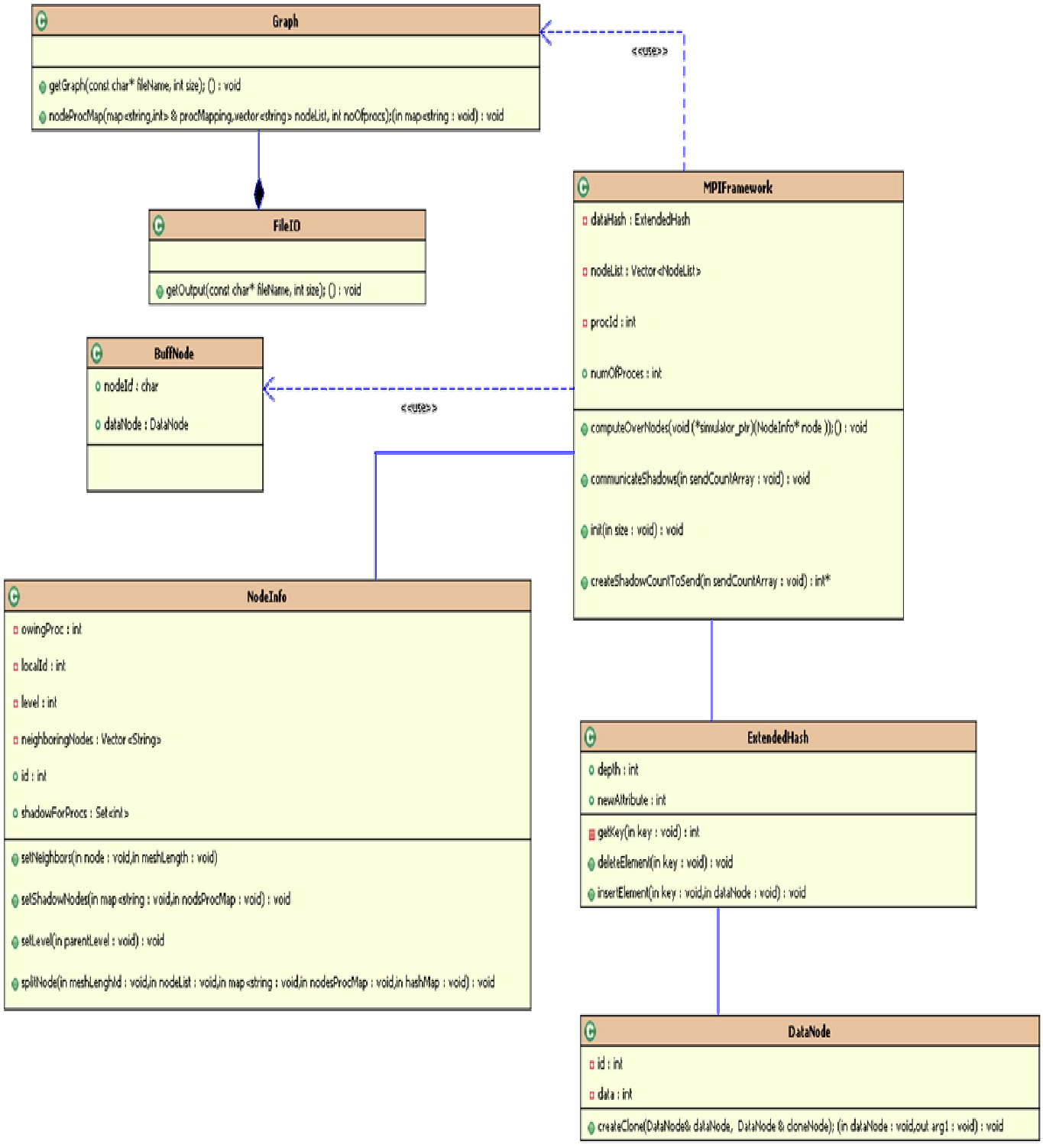
[34] *Clinton P. T. Groth*, A Parallel Adaptive Mesh Refinement (AMR) Computational Framework for Physically Complex Flows, University of Toronto Institute for Aerospace Studies, MITACS & Fields Aeronautics Workshop, April 28-29, 2005

[35] http://seesar.lbl.gov/ANAG/chombo

[36] http://kbingham.net/polytropes.htm

[37] www.clawpack.org

APPENDIX A CLASS DIAGRAM OF THE FRAMEWORK



APPENDIX B DAPTIVE MESH FRAMEWORK CODE

This file includes the definition used in the framework.

Define.h

```
#include <iostream>

#define MAX_NEIGHBORS 4

#define SHADOWS_FOR_PROCS 4

#define CHILD_NODES_WIDTH 2

#define MAX_SIZE_FOR_RECVBUFFER 6

#define NAME_LENGTH 20

#define LOAD_LIMIT 4
```

This file includes the class definition of NodeInfo class. This class holds the node information of the application node.

Nodeinfo.h

```
#ifndef NODE_INFO_H

#define NODE_INFO_H


#include <iostream>

#include "define.h"

#include <vector>

#include "extendedHash.h"

#include "node.h"

#include <map>
```

```
#include <set>


using namespace std;


/* Node information common for every application*/
class NodeInfo
{
private:
        int owingProc;
        int localId;
        int level;
        vector<string> neighboringNodes;



public:
        string id;
        set<int> shadowForProcs;


        NodeInfo(string globalId,int localId,int procID);
        ~NodeInfo();
        void setNeighbors(NodeInfo& node,int meshLength = 2);
        void setShadowNodes(map<string,int> nodsProcMap);
        void setLevel(int parentLevel);
```

```
        void splitNode(int meshLength, vector<NodeInfo>& nodeList,map<string,int>
nodesProcMap, ExtendedHash& hashMap);
    };

    #endif
```

This file includes the implementation of NodeInfo class

nodeInfo.cpp

```
#include <iostream>
#include <string>
#include "nodeInfo.h"
#include <map>
#include <sstream>

using namespace std;

NodeInfo::NodeInfo(string globalId,int localId,int procID)
{
        this->owingProc = procID;
        this->id = globalId;
        this->localId = localId;
        this->level =0;
```

```
}

NodeInfo::~NodeInfo()
{
}

void NodeInfo::setLevel(int parentLevel)
{
        level = parentLevel++;
}

void NodeInfo::splitNode(int meshLength,vector<NodeInfo> & nodeList, map<string,int>
nodesProcMap,ExtendedHash& hashTable)
{
        vector <NodeInfo>::iterator Iter;
        int position=0;
        DataNode* dataNode = hashTable.getElement(this->id);

        //iterate to find the location of the parent nodeInfo
        for ( Iter = nodeList.begin( ) ; Iter != nodeList.end( ) ; Iter++ )
        {
                if((*Iter).id == this->id)
                {
```

```
                nodeList.erase(Iter);
                break;
        }
        position++;
}

// Create new 4 nodes
for(int j=0; j<MAX_NEIGHBORS; j++)
{
        char* str;
        string globalId = id+","+itoa(j,str,10);
        NodeInfo node = NodeInfo(globalId,j,this->owingProc);
        node.setLevel(this->level);
        node.setNeighbors(node,meshLength);
        nodesProcMap[globalId] = this->owingProc;
        DataNode childNode;
        childNode.createClone(*dataNode , childNode);
        hashTable.insertElement(node.id,&childNode);
        // consider the order of adding the nodes to the vector
        nodeList.insert(nodeList.begin()+position,node);
        position++;
}
position - 3;
```

```
        for(Iter = nodeList.begin()+position; Iter< nodeList.begin()+4; Iter++)
        {
                (*Iter).setShadowNodes(nodesProcMap);

        }
        hashTable.deleteElement(this->id);


}

void NodeInfo::setNeighbors(NodeInfo& node, int meshLength)
{
        int localid;

        if(node.level == 0)
        {
                localid = atoi(this->id.c_str());
                if((localId-meshLength) >0)
                {
                        stringstream out;
                        localid =localid-meshLength;
                        out<<localid;
                        this->neighboringNodes.push_back(out.str());
```

```
}
else
        this->neighboringNodes.push_back("");

if(((localId+1)%meshLength )>0)
{       stringstream out;
        out<<localid+1;
        this->neighboringNodes.push_back(out.str());
}
else
        this->neighboringNodes.push_back("");

if(((localId-1)%meshLength )==0)
{
        stringstream out;
        localid =localid-1;
        out<<localid;
        this->neighboringNodes.push_back(out.str());
}
else
        this->neighboringNodes.push_back("");

if((localId+meshLength)<meshLength*meshLength)
```

```
                {
                        stringstream out;
                        localid = localid+meshLength;
                        out<<localid;
                        this->neighboringNodes.push_back(out.str());
                }
                else
                        this->neighboringNodes.push_back("");
        }
        else
        {
                localid = node.localId;
                switch (localId)
                {
                case 0:
                        {
                                stringstream out1, out2;
                                node.neighboringNodes.push_back(this-
>neighboringNodes[0]);
                                out1<<localid +1;
                                node.neighboringNodes.push_back(out1.str());
                                out2<<localId+ CHILD_NODES_WIDTH;
                                node.neighboringNodes.push_back(out2.str());
```

```
                    node.neighboringNodes.push_back(this-
>neighboringNodes[3]);
                    break;
                }
            case 1:
                {
                    stringstream out1, out2;
                    node.neighboringNodes.push_back(this-
>neighboringNodes[0]);
                    node.neighboringNodes.push_back(this-
>neighboringNodes[1]);
                    out1<<localId +CHILD_NODES_WIDTH;
                    node.neighboringNodes.push_back(out1.str());
                    out2<<localId -1;
                    node.neighboringNodes.push_back(out2.str());
                    break;
                }
            case 2:
                {
                    stringstream out1, out2;
                    out1<<localId -CHILD_NODES_WIDTH;
                    node.neighboringNodes.push_back(out1.str());
                    out2<<localId +1;
```

```
                    node.neighboringNodes.push_back(out2.str());
                    node.neighboringNodes.push_back(this-
>neighboringNodes[2]);
                    node.neighboringNodes.push_back(this-
>neighboringNodes[3]);
                    break;
                }
            case 3:
                {
                    stringstream out1, out2;
                    out1<<localId -CHILD_NODES_WIDTH;
                    node.neighboringNodes.push_back(out1.str());
                    node.neighboringNodes.push_back(this-
>neighboringNodes[1]);
                    node.neighboringNodes.push_back(this-
>neighboringNodes[2]);
                    out2<<localId -1;
                    node.neighboringNodes.push_back(out2.str());
                    break;

                }
            }
        }
```

```
}

void NodeInfo::setShadowNodes(map<string, int> nodesProcMap)
{

        for(int i=0; i<this->neighboringNodes.size(); i++)
        {

                if(nodesProcMap[this->neighboringNodes[i]]!=this->owingProc)
                {
                        this->shadowForProcs.insert(nodesProcMap[this-
>neighboringNodes[i]]);
                }
        }
}
```

This class defines the application node data structure. User should supply this class and its implementation.

Node.h

```
#ifndef NODE_H
#define NODE_H
```

```
#include <iostream>
#include "define.h"
#include <vector>



using namespace std;

/* User application defined data node class */
class DataNode
{
private:
        int id;
        int data;

public:
        DataNode();
        ~DataNode();
        DataNode* createClone(DataNode& dataNode,  DataNode & cloneNode);
};



class BuffNode
{
```

```
public:
        char nodeId[NAME_LENGTH];
        DataNode dataNode;

        BuffNode();
        ~BuffNode();
};

#endif
```

This file is the implementation of the DataNode class

dataNode.cpp

```
#include <iostream>
#include "node.h"

DataNode::DataNode()
{
}

DataNode::~DataNode()
{
```

```
}

DataNode* DataNode::createClone(DataNode & dataNode, DataNode & cloneNode)
{
	cloneNode.id = dataNode.id;
	cloneNode.data = dataNode.data;
	return &cloneNode;
}

BuffNode::BuffNode()
{
	sprintf(nodeId,"");
}

BuffNode::~BuffNode()
{
}
```

Extended hash is defined in this file.

extendedHash.h

```
#ifndef EXTENDEDHASH_H
#define EXTENDEDHASH_H
```

```
#include <iostream>
#include <map>
#include "node.h"
#include <string>



using namespace std;



class ExtendedHash
{
private:
	int depth;
	map<string, DataNode*> hashMap;
	int getKey(char* key);

public:
	ExtendedHash();
	~ExtendedHash();
	DataNode * getElement(string key);
	void deleteElement(string key);
	void insertElement(string key,DataNode* dataNode);
};
```

```
#endif
```

Extended hash implementation is done in this file

extendedHash.cpp

```
#include <iostream>
#include "extendedHash.h"


using namespace std;

ExtendedHash::ExtendedHash()
{
}

ExtendedHash::~ExtendedHash()
{

}

DataNode * ExtendedHash::getElement(string key)
{
	return hashMap[key];
}
void ExtendedHash::deleteElement(string key)
{
	delete(hashMap[key]);

}

void ExtendedHash::insertElement(string key,DataNode* node)
{
	hashMap[key]=node;
}
```

```
int ExtendedHash::getKey(char* key)
{
	///*int first,second,third;
	//
	//first = atoi(strtok(key,","));
	//second = atoi(strtok(NULL,","));
	//t*/hird = atoi(strtok(NULL,","));

	return 0;
}
```

Initial file handling is done in FileIO class and it is defined in following file.

fileIo.h

```
#ifndef FILEIO_H

#define FILEIO_H



#include <iostream>

#include <fstream>

#include <string>

#include <vector>



using namespace std;



class FileIO

{

public:

	FileIO();
```

```
        ~FileIO();
        vector<string> getOutput(const char* fileName, int size);
};
#endif
```

Implementation of the FileIO is done here.

fileIO.cpp

```
#include <iostream>
#include <string>
#include "fileIO.h"

using namespace std;

FileIO::FileIO()
{
}

FileIO::~FileIO()
{
}
vector<string> FileIO::getOutput(const char* fileName, int size)
```

```
{
        string line;
        int i = 0;
        vector<string> strVect;
        char** stringArray;
        ifstream file(fileName);
        if(!file)
        {
                cerr << "Unable to open file "<< fileName <<endl;
                exit(1);
        }
        if(file.is_open())
        {
                cout<<endl<<"printing the text file"<<endl;
                while(!file.eof())
                {
                        getline(file,line);
                        strVect.push_back(line);
                        cout<<strVect[i]<<endl;
                        i++;
                }
                file.close();
        }
```

```
        return strVect;
}
```

Graph handling is done in following file.

Graph.h

```
#ifndef GRAPH_H
#define GRAPH_H

#include <iostream>
#include <vector>
#include <map>
#include <string>


using namespace std;

class Graph
{

public:
        Graph();
        ~Graph();
```

```
        vector<string> getGraph(const char* fileName, int size);
        /*how the redistribution take place? */
        void nodeProcMap(map<string,int> & procMapping,vector<string> nodeList, int
noOfprocs);



};

#endif
```

Implementation of the Graph is done in following file.

Graph.cpp

```
#include <iostream>
#include <vector>
#include "graph.h"
#include "fileIO.h"
#include <map>
#include <cmath>

using namespace std;

Graph::Graph()
{
```

```
}

Graph::~Graph()
{
}

vector<string> Graph::getGraph(const char* fileName,int size)
{
	FileIO fileio;
	int arraysize;
	arraysize = size*size;
	vector<string> output;
	int i=0, j=0 , k=0;
	vector<string> graphVector;
	vector< vector<int>> graphArray(size,size);

	for (i =0; i<size; i++)
	{
		for( j=0; j<size; j++)
		{
			graphArray[i][j] = k;
			k++;
		}
```

```
        }
        char * str = new char();
        output = fileio.getOutput(fileName,arraysize);
        cout<<"output to graph"<<endl;
        for(i=0; i< arraysize;i++)
        {
                j=atoi(output[i].substr(0,int(output[i].find(","))).c_str());
                k=atoi(output[i].substr(int(output[i].find(",")+1)).c_str());
                graphVector.push_back(itoa(graphArray[j][k],str,10));
        }
        return graphVector;
}

void Graph::nodeProcMap(std::map<string,int> &procMapping,vector<string> nodeList, int
noOfprocs)
{
        double nodeSize;
        nodeSize = (double)nodeList.size();
        int noOfItems =ceil(nodeSize/noOfprocs);
        int i,j,k;
        j=0;
        k=0;
        for(i=0;i<nodeList.size();i++)
```

```
        {
                if(j<noOfItems)
                {
                        procMapping[nodeList[i]]= k;
                        j++;
                }
                else
                {
                        k++;
                        procMapping[nodeList[i]]= k;
                        j=1;
                }
        }
}
```

Framework.h

```
#ifndef FRAMEWORK_H
#define FRAMEWORK_H

#include <iostream>
#include "graph.h"
#include <vector>
#include <string>
```

```
#include "nodeInfo.h"
#include "extendedHash.h"
#include "node.h"
#include "define.h"

using namespace std;

class MPIFramework
{
private:
        ExtendedHash dataHash;
        vector<NodeInfo> nodeList;
        int procID;

public:
        int numOfProcs;
        MPIFramework(int size);
        ~MPIFramework();
        void computeOverNodes(void (*simulator_ptr)(NodeInfo* node ));
        void communicateShadows(int* sendCountArray);
        void init(int size);
        int* createShadowCountToSend(int* sendCountArray);
        void MPIFramework::loadBalance();
```

```
};

#endif

#include <iostream>
#include "mpiframework.h"
#include <cmath>



using namespace std;

MPIFramework::MPIFramework(int size)
{
        init(size);
}

MPIFramework::~MPIFramework()
{
}

void MPIFramework::init(int size)
{
        char cmmd[40];
```

```
//string comm;
int meshLength;

this->procID=0;
this->numOfProcs =4;
meshLength = size;
/*MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&procID);
MPI_Comm_size(MPI_COMM_WORLD,&num_procs);

MPI_Barrier(MPI_COMM_WORLD);
time_elapsed = -MPI_Wtime();*/

/* Invoking hilbert curve for the initial node orientation of the mesh*/
if (procID == 0 && numOfProcs > 1)
{
        /*sprintf(hilbert_str,"./hilbert %d",size);
        printf("hilbert str=%s",hilbert_str);*/
        sprintf(cmmd,"hilbert.exe %d ",size);
        int i = system(cmmd);
}

/* Create the list of nodes generated from hilbert's curve*/
```

```
	Graph graph;
	vector<string> graphList;
	graphList = graph.getGraph("output.txt",size);
	std::map<string,int> nodeProcMap;
	graph.nodeProcMap(nodeProcMap,graphList,numOfProcs);

	for(int i =0; i<graphList.size();i++)
	{
		if(nodeProcMap[graphList[i]] == procID)
		{
			NodeInfo node = NodeInfo(graphList[i],atoi(graphList[i].c_str()),this-
>procID);
			node.setLevel(0);
			node.setNeighbors(node,meshLength);
			node.setShadowNodes(nodeProcMap);
			this->nodeList.push_back(node);
		}
	}

		/* Create the Data node list */
	for(int i =0; i< graphList.size(); i++)
	{
```

```
                DataNode* dataNode = new DataNode();
                dataHash.insertElement(graphList[i],dataNode);
        }
}

void MPIFramework::computeOverNodes(void (*simulator_ptr)(NodeInfo* ))
{

        vector<NodeInfo>::iterator Iter;
        for(Iter = this->nodeList.begin(); Iter <nodeList.end(); Iter++)
        {
                (*simulator_ptr)(&(*Iter));
        }
}

void MPIFramework::communicateShadows(int* sendCountArray)
{
        BuffNode **recvbuffer_arr,** sendbuffer_arr, buff;
        int i,j,k =0;
        int* objCount = new int[numOfProcs];

        //int blockcounts[2];
        //MPI::Datatype buffer_datatype, oldtypes[2];
```

```
//MPI::Aint offsets[2];
//MPI::Status status;
//MPI::Request pending;

//bufcounts[0]=20;
//bufcounts[1]=1;

//offsets[0] = 0;
//offsets[1] = (int)&buff.nodeId - (int)&buff;

//oldtypes[0]=MPI::CHAR;
//oldtypes[1]=DataNode;

//          /* Define structured type & commit it...*/
//MPI_Type_struct(2,bufcounts,offsets,oldtypes,&buffer_datatype);
//MPI_Type_commit(&buffer_datatype);
//

recvbuffer_arr = new BuffNode*[numOfProcs];
sendbuffer_arr = new BuffNode*[numOfProcs];
/* Allocate memory for recvbuffers...*/
for (i=0;i<this->numOfProcs;i++)
{
```

```
        if (sendCountArray[i]!=0)
        {
                recvbuffer_arr[i] = new BuffNode[MAX_SIZE_FOR_RECVBUFFER];
                sendbuffer_arr[i] = new BuffNode[sendCountArray[i]];
        }
        else
        {
                /* get exception*/
                recvbuffer_arr[i]=NULL;
                sendbuffer_arr[i]=NULL;
        }
}

for(i=0;i<numOfProcs;i++)
        objCount[i] = 0;




set<int>::iterator itr;
/* send shadow nodes data*/
for(i=0; i<this->nodeList.size(); i++)
{
```

```
                for(itr= nodeList[i].shadowForProcs.begin(); itr!=
nodeList[i].shadowForProcs.end(); itr++)
                {
                        strcpy(sendbuffer_arr[*itr][objCount[*itr]].nodeId,nodeList[i].id.c_str());
                        sendbuffer_arr[*itr][k].dataNode = *(this-
>dataHash.getElement(nodeList[i].id));
                        objCount[*itr]++;
                }
        }

        // Recv buffers...
        for (i=0;i<numOfProcs;i++)
        {
                /*if (sendCountArray[i]!=0)
                {
                        MPI::send();

        MPI::Recv(recvbuffer_arr[i],MAX_SIZE_FOR_RECVBUFFER,buffer_datatype,i,1,com
m,&status);
                }*/
        }
        /* unpack the buffer array */
```

```
	/* delete allocated space*/
	for(i=0; i<this->numOfProcs;i++)
	{
		delete []recvbuffer_arr[i];
	}
	delete [] recvbuffer_arr;

}

/*x?"True":"False")*/




int* MPIFramework::createShadowCountToSend(int* sendCountArray)
{
	vector<NodeInfo>::iterator Iter;
	for(int i=0; i<this->numOfProcs; i++)
	{
		sendCountArray[i] = 0;
		//cout<<sendCountArray[i];
	}
	set<int>::iterator itr;
```

```
        for(Iter= nodeList.begin(); Iter<nodeList.end(); Iter++)
        {
                /* travers each shadow proc to find the no of elements to send*/
                for(itr =(*Iter).shadowForProcs.begin(); itr!=(*Iter).shadowForProcs.end(); itr++)
                {
                        sendCountArray[*itr]++;
                }
        }
        return sendCountArray;
}

void MPIFramework::loadBalance()
{
/*proc 0 get nodes from each procs.
put them together and divide equally.
then bcast boundary to all procs
each proc will prepare for send and recieve of nodes from neighbors*/

        int * nodesCount = new int[numOfProcs];
        int i,totSize,avgCount,rem,totBefore;
        bool unbalance = false;

        for(i=0; i<numOfProcs;i++)
```

```
        nodesCount[i] =0;


int size = this->nodeList.size();
nodesCount[this->procID] =size;


//MPI_Bcast(size);
/* receive nodesize from each processor*/
//MPI_Recv();


for(i=0; i<numOfProcs; i++)
        totSize +=nodesCount[i];



avgCount = floor((double)totSize/numOfProcs);
rem = totSize - (avgCount * numOfProcs);


for(i=0; i<numOfProcs; i++)
{
        if(nodesCount[i]-avgCount > LOAD_LIMIT)
        {
                unbalance =true;
                break;
        }
```

```
}

if(unbalance)
{
        totBefore =0;
        for(i=0; i<this->procID; i++)
        {
                if(rem >0)
                {
                        if(nodesCount[i] - avgCount > 0)
                        {
                                rem--;
                                totBefore += avgCount + 1;
                        }
                        else
                                totBefore += avgCount;
                }
                else
                        totBefore += avgCount;
        }


}
```

```
}
```

Main.cpp

```
#include <iostream>
#include "mpiframework.h"
#include <math.h>

using namespace std;

void SimulatorFunction(NodeInfo *);

int main(int argc, char *argv[])
{
	int* sendCountArray;
	int width = -1;
	double p =0.0;
```

```
while (width < 2) {
        cout<<"Enter the width of mesh in number of nodes."<<endl;
        cin>>width;

        if (width<0)
        {
                // exit(0);
                 return -1;
        }

        p = (log10((double)width)/log10((double)2));/* Check width is result of 2^m*/
        if (p != ((int)p)) {
                cout<<"Mesh width must be >= 2, and the result of 2^m (m =
1,2,3,4...)"<<endl;
                width = -1;
        }
}
/*MPI::Init(&argc,&argv);

MPI_Barrier(MPI_COMM_WORLD);

time_elapsed = -MPI_Wtime();*/
```

```
	MPIFramework framework = MPIFramework(width);
	sendCountArray = new int[framework.numOfProcs];
	for(int i =0 ; i<5; i++)
	{
		sendCountArray = framework.createShadowCountToSend(sendCountArray);
		for(int i=0; i<4; i++){
			cout<<sendCountArray[i];
		}
		/* create a class to send and receive shadow nodes and fill them here*/
		framework.computeOverNodes(SimulatorFunction);
		/*framework class know where the shadow nodes coming from*/
		framework.communicateShadows(sendCountArray);
	}

	//MPI::Finalize();
	cin.get() ;
	return 0;
};

void SimulatorFunction(NodeInfo* node)
{
	/*do some work here*/
```

```
        cout<<" simulator function was called"<<endl;
}
```