AN ADAPTIVE MESH REFINEMENT (AMR) LIBRARY USING CHARM++

BY

PUNEET NARULA

B.Engr., University of Roorkee, 1999

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

To my Family.

Acknowledgments

First and foremost, I would like to express my gratitude to my advisor, Professor L. V. Kale. His generous support, guidance, encouragement and understanding enabled me to carry out this work successfully. I would also like to thank Orion Lawlor who wrote the serial version of the benchmark and helped in the editing of this thesis.

A special thanks to all my friends who have been an invaluable source of encouragement and enthusiasm. I would also like to thank the members of PPL for many stimulating discussions and maintaining a pleasant working environment. Last but not the least I would like to thank Dominique Kilman for all her help and support throughout the writing of this thesis.

Table of Contents

Chapter 1 Introduction **1**
1.1 Why Adaptive Mesh Refinement? 2
1.2 Adaptive Mesh Refinement Library Objectives 3
1.3 Thesis Organization 3

Chapter 2 Adaptive Mesh Refinement and Related Work **4**
2.1 AMR 4
2.2 Constituents of AMR 5
2.2.1 Error Estimation 5
2.2.2 Domain Decomposition 5
2.2.3 Parallelization - Parallel Data Structure Implementation 6
2.2.4 Communication 7
2.3 Other Related Work 7

Chapter 3 Charm++ and Parallel Objects **9**
3.1 Libraries Using Charm++ 11
3.2 The Pack/Unpack Framework 12

Chapter 4 Load Balancing **18**
4.1 Load Balancing Strategies 18
4.1.1 Centralized Load Balancing 19
4.1.2 Neighborhood Load Balancing: NeighborLB 20
4.2 Load Balancing Strategies API 21

Chapter 5 Adaptive Mesh Refinement Library **23**
5.1 Introduction 24
5.2 Library's User Interface 27
5.2.1 Creation of Library 27
5.2.2 User Data Class 27
5.2.3 Load Balancing 31
5.2.4 Flow of Execution for the Library 31
5.3 Library Design 32
5.3.1 Hierarchical Indexing of Tree 32
5.3.2 Neighbor Communication 35
5.3.3 Class Design 39

5.4 Preliminary Performance Measurement . . . 41
5.4.1 Benchmark Application . . . 41
5.4.2 Results . . . 41

Chapter 6 Conclusions . . . 44
6.1 Future Work . . . 44

References . . . 46

Simple 2D AMR Apllication . . . 49
.1 Writing the .ci File . . . 49
.2 Writing the .h File . . . 49
.3 Writing The .C File . . . 52

List of Figures

1.1 Typical Grid for 2D AMR 2

3.1 A simple class declaration showing the `pup` method 15
3.2 Packing and unpacking a *foo* object 16
3.3 Declaration of the *bar* class showing the `pup` method. 17

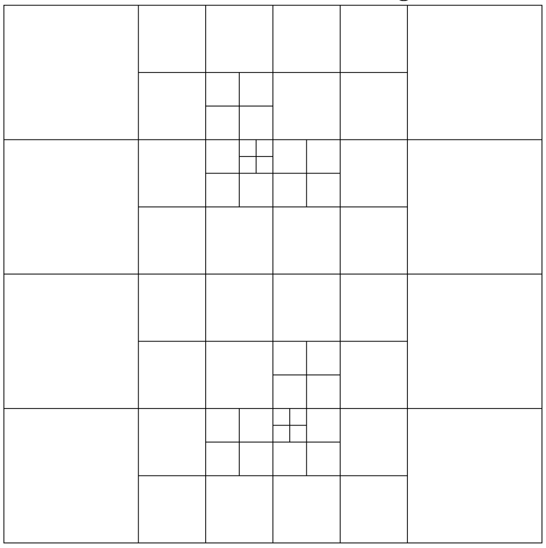
5.1 Mapping of grid to the quad tree in 2D AMR 23
5.2 Typical algorithm for application with nearest neighbor communication pattern 24
5.3 Basic algorithm being executed in parallel 25
5.4 Creating the library from the mainchare 26
5.5 Message Splitting for 2D AMR 30
5.6 Order of creation of objects at the begining of execution for 2D AMR 31
5.7 Indexing in 2D AMR (quad Tree) 35
5.8 Nearest Neighbors for the leaf 36
5.9 Neighbor Communication:Sender and Receiver at the same level in a quad tree 37
5.10 Neighbor Communication:Sender at a coarser level than the Receiver in a quad tree 38
5.11 Neighbor Communication: Sender at a finer level than the Receiver in a quad tree 39
5.12 Class Diagram for the AMR Library 40
5.13 Performance Graph without load balancing 42
5.14 Performance Graph with heapCentral load balancing 42
5.15 Performance Graph with refine load balancing 42

Chapter 1

Introduction

Parallel computing has enjoyed considerable growth in recent years and parallel architectures are assuming an increasingly central role in information processing [7]. The stimulus behind this growth has been the ever increasing demand for compute power to solve complex problems in the shortest possible time. This sustained growth has also manifested itself in the evolution of a diverse design space for parallel machines. In order to exploit the benefits offered by the multitude of architectures available it is important for parallel applications to be developed in a machine independent fashion making only the most generally applicable assumptions about the underlying architecture. This requirement has been addressed by the developement of the *Converse* parallel runtime system and the *Charm++* parallel programming language. Though the growth in the field of parallel computing has been tremendous, it has not been optimal because effort is wasted in duplicating the capabilities required for parallel computing. Thus there is a need for writing more reusable libraries and modules, which would enhance the growth in the area manyfold. This thesis is an effort in this direction. It provides a library for the *Adaptive Mesh Refinement* (AMR) technique which is used in a variety of applications like cosmology, global atmospheric modelling, hyperbolic partial differential equations[2], shock hydrodynamics [1], etc. The library described in this thesis facilitates the implementation of applications using 1D, 2D and 3D AMR in parallel.

Figure 1.1: Typical Grid for 2D AMR



1.1 Why Adaptive Mesh Refinement?

Adaptive Mesh Refinement, AMR, is a technique used to solve partial differential equations on structured grids. In this technique the areas of more interest are refined. Solutions without AMR are less efficient because the whole mesh is evenly refined, rather than refining only the most interesting parts, thus wasting computer resources. AMR applications typically require adaptation, at runtime, of only the areas where most of the small scale phenomenon occur. Though the potential savings is significant with AMR, it is not used as frequently as it should be, because of the complexity of implementation. Also, parallelizing an AMR application presents significant challenges because of the coding complexity and performance issues. Hence the motivation for a parallel library to abstract the parallel data structures and give a cleaner interface to work with. Due to the dynamic nature of AMR, load imbalance is a problem which becomes a bottleneck in obtaining speedups. Since the AMR problem is dynamic, the load balancing has to be done at runtime as there is no way to get a good load distribution statically. Figure 1.1 shows a typical refined grid for 2D AMR.

1.2 Adaptive Mesh Refinement Library Objectives

In this thesis we describe the design and implementation of an *Adaptive Mesh Refinement* library using *Charm++*[16][13][14][15], an object oriented parallel programming language based on C++, and Converse[11], a parallel runtime system that enables parallel implementations of a variety of programming languages and paradigms and also facilitates interoperability among various paradigms. Typically an AMR application is composed of four basic parts:

- Error Estimation Function to determine when to refine
- Domain Decomposition
- Parallel data structure and data distribution over processors
- Data communication

This library abstracts away the later two parts of AMR application development from the user. The *Charm++* system provides dynamic load balancing[4][5][6], so that the user can use different strategies (described in chapter 4) with less coding overhead. The AMR library can be obtained with the current distribution of *Charm++*.

1.3 Thesis Organization

The thesis consists of 6 chapters. Chapter 2 discusses Adaptive Mesh Refinement techniques in general, and presents an overview of related work in the field. Chapters 3 and 4 describe the issues related to the language and load balancing respectively. They have the basic information required about Charm++ and the load balancing framework. Chapter 5 describes the design of the AMR library and various design decisions taken. It also describes the API for the library and gives the preliminary performance results of the library. Conclusions and future work are discussed in Chapter 6.

Chapter 2

Adaptive Mesh Refinement and Related Work

2.1 AMR

Adaptive mesh refinement techniques have been shown to be very successful in reducing the computational and storage requirements for solving partial differential equations[22]. Rather than using a uniform mesh with grid points evenly spaced in a domain, adaptive mesh refinement techniques place more grid points in areas where the local error in the solution is large. The mesh is adaptively refined and/or unrefined during the computation according to local error estimates on the domain. This technique is much more efficient then the use of uniform meshes when the solution changes more rapidly in some areas than others. Typically AMR comes in two flavours[2]:

- Dynamic Gridding - In this variant of AMR the nodes are moved in the time-space domain continuously to where they are needed the most. This technique is also called mesh movement or r-refinement.

- Static Gridding - In this technique the basic grid is overlaid by finer grids where the accuracy of the solution needs to be improved. The grid is updated only at discrete time steps. This technique is also called mesh enrichment or h-refinement.

We use the later technique of static gridding as it is a more accepted solution for finite difference schemes[2]. In this chapter we will discuss the various parts needed for developing an AMR application, related work that has been done in this field and the motivation behind writing the library.

2.2 Constituents of AMR

In order to design any library it is important to identify the parts of a typical application, and determine which parts are to be implemented by the library and which parts should be left to the user.

2.2.1 Error Estimation

One of the most important tasks in any AMR application is the error estimation scheme. The decision to refine is based on the local error estimate. The importance of this cannot be stressed more and a lot of research has gone into this area. This is a non trivial problem because without the analytical solution to the problem at hand, it is hard to determine the error in the calculated approximate solution. A true error estimation function would be an ideal choice for this task, but such a function is very expensive. Many heuristic solution's exist to tackle the problem, which have proven useful. Berger and Collela describe one such method in [1]. [29] describes a heuristic error estimation function based on the curvature of the solution. This problem is typically the job of numerical scientists and we do not attempt to solve this. We leave this for the user of the library to implement, as we feel that they are in a better position to make a decision based on the problem domain.

2.2.2 Domain Decomposition

This is another important topic in the AMR community with people differing in their ideas of how to tackle this problem. Essentially, there are two approaches, and each has its merits. The first one was given by Marsha Berger [2] with the central idea to fit rectangular subgrids over clusters of flagged points. Points which are determined to have local error greater than the threshold are flagged. In this scheme the grids can overlap and can be rotated. The second approach was given by Trompert and Verwer in [29] which avoids the domain decomposition alltogether. Instead they simplify the process by refining all the flagged points. But in turn they have a complicated approach to store subgrids which is hard to parallelize. The first method has inherent paralelism and uses uniform grids which allows for faster code for integrators but has a slow clustering phase and unnecessarily refines some cells. The second method has a very simple refinement criteria and does not refine any unnecessary cells which saves memory but is not inherently parallel and the communication patterns are highly irregular as there are no clearly defined subgrids. [28] discusses the pros and cons of each approach and concludes that having a hybrid scheme which takes advantages of both the approaches will be the step in the right direction.

Our approach is not to use domain decomposition and instead use the simplistic approach of flag and refine. But we choose uniform subgrids in an appropriate data structure which can be easily parallelized. Though the load is highly irregular, we take advantage of the dynamic load balancing capabilities of Charm++ (described in Chapter 4).

2.2.3 Parallelization - Parallel Data Structure Implementation

The primary data structure for AMR is a tree (oct tree for 3D and quad tree for 2D). It is the responsibility of the library to implement a tree distributed over multiple processors. It is important to be able to easily add and delete nodes from this tree. The leaves in the tree should be able to talk to their neighbors despite the changing structure of the tree because of refinement. Since we want to be able to run the code on distributed memory machines or clusters, we cannot use pointers to keep track of the neighbors. Instead we design the indexing scheme of the tree in such a way that a leaf can determine its neighbors by itself. Chapter 5 descibes this in greater detail. Another important aspect that comes forward because of parallelization of the problem is load balancing. It is a more challenging problem because of the continuous changes in the tree structure due to refinement. It is important to note that we adopt a modular approach and do not couple the amr library with load balancing so that we can use different strategies with different problems. [19][4] discuss some schemes which have been suggested to do dynamic load balancing.

2.2.4 Communication

With processors becoming faster day by day, one cannot ignore the communication cost if good speedups are desired. As in the case of parallel data strctures, it is the task of the library to hide the implementation details of communication from the user. The *Charm++* system provides communication optimization as discussed in [30]. At the library level we make sure that instead of using all to one communication we use reduction which helps in reducing the communication latency.

2.3 Other Related Work

In the recent past many people have written AMR libraries like DAGH[23], PARAMESH[21], AMRA[26], SAMRAI[20] etc. The problem with most of these libraries is that they have been developed in Fortran. Fortran was the language of choice for numerical computing because it was substantially faster than object oriented languages like C++. Though Fortran was fast, it did not offer software features like polymorphism etc., that promotes the reuse of code. The basic challenge that all the library developers in numerical computing face is the tradeoff between the speed of the code and good object oriented software techniques. In the past, the difference between the speed of execution of the code in Fortran and C++ was tremendous, but with the evolution of better compilers for C++ this gap has been bridged to a great extent. More recently there is a shift in the direction of using object oriented languages like C++ for numerical computing. SAMRAI is one of frameworks that was developed in C++.

They have made extensive use of patterns and software techniques during the development of their framework[10]. Our library also is an effort in that direction. The library presented in this thesis is modular, so components can be changed independently without affecting the others. Also the library does not make any assumption about the user's data structure in each node. Most of the libraries do not allow arbitrary data structures inside the nodes thus limiting the potential of the library. Though the implementation provided is only for 1D, 2D and 3D, the design is general enough to allow the extension of the library to higher dimensions.

Chapter 3

Charm++ and Parallel Objects

Charm++ is an object oriented parallel programming system based on C++. It is explicitly parallel in nature and provides a clear separation between sequential and parallel objects. The execution model of Charm++ is message driven; a Charm++ program in execution is a collection of concurrent objects of various types (described in more detail later) that communicate by sending messages to one another [16]. Charm++ programs can freely use most object-oriented and generic programming features of C++, including multiple inheritance, polymorphism, overloading, strong typing and templates.

Every Charm++ object is a regular C++ object, i.e. an instance of a C++ class. Concurrent and replicated objects in Charm++ have several special attributes unique to the objects that are provided and supported by the run-time system.

A Charm++ object belongs in one of the following categories:

- Concurrent Objects
  - Chares - Chares are the most important entities in a Charm++ program. Unlike ordinary C++ objects, Chares can be created asynchronously on remote processors and special methods, called *entry methods* on these objects may be invoked asynchronously from remote processors. An entry method is invoked by sending a message to the Chare.
  - Chare Groups - Chare groups are a special type of concurrent object. Each chare

group is a collection of chares with one branch on every processor. All members of a chare group share a globally unique identifier and messages may be broadcast to the whole group or to a specific branch.

  - Chare Nodegroups - Chare Nodegroups are similar to groups, except that instead of having one branch on every processor, nodegroups have one branch on every SMP node that the program runs on.
  - Chare Arrays - Chare Arrays[18] are generalized collections of Chares, however these collections are not constrained by the underlying architecture. Chare arrays can have any number of elements and this number may change at runtime. These objects can migrate and hence can be load balanced both at runtime and statically. Chare Arrays support sparse arrays and arbitrary indexing of arrays which makes it a versatile and flexible data structure, which can be used in variety of applications with very little overhead.

- Sequential objects - These are regular C++ objects, except that they may not have static data members. These objects are local to a processor and the Charm++ runtime environment has no knowledge of their existence. They are typically members of other concurrent objects.

- Messages (Communication objects) - These are entities that constitute the arguments to asynchronously invoked methods of concurrent and replicated objects. A Charm++ message consists of an *envelope* followed by the message body. The envelope is used by the Charm++ runtime environment and stores message attributes such as the message type, source processor, etc. Entry methods that handle the messages deal only with the message body.

- Readonly objects - A readonly object represents a global variable in the computation. Charm++ does not allow mutable global variables in the computation in order to keep programs portable across a wide range of platforms. Readonly objects are a way to share data amongst all the objects involved in a computation. In cases where the size of the readonly object is not known at compile time, readonly *messages* may be used. Readonly messages are declared as pointers. Pointers to messages are the only readonly pointers allowed.

3.1 Libraries Using Charm++

In traditional libraries in message passing environments like MPI[8], library computations are invoked by regular function calls. The library call blocks the caller on all processors. After completion, the library module returns the result of the call and control to the calling modules. Some disadvantages of libraries in this style are:

- Idle times in the library computation cannot be utilized even if there are other independent computations
- Caller modules must invoke the library on all processors even when only a subset of the processors provide input, and receive output.
- Library computations must be called in the same sequence on every processor.

A message-driven system, such as *Charm++*, supports multiple objects per processor and uses a pool of messages on every processor. An object is scheduled for execution when there is a message to be delivered to it. In such a system, one can invoke multiple library modules concurrently, allowing them to naturally overlap their idle times with useful computations. This is a substantial boost for encouraging use of libraries. [12] discusses, in detail, the advantages of writing libraries using the message-driven paradigm used in *Charm++*. The other advantage of using *Charm++* is that it provides a load balancing framework[4] for dynamic load balancing of Chare Arrays. The dynamic load balancing uses the migration support provided by the PUP framework (discusssed in the next section) to do the load balancing.

3.2 The Pack/Unpack Framework

The pack/unpack or "pup" framework is a collection of efficient and elegant classes that enable the Chare Arrays of Charm++ to be migrated from one processor to another processor or to the disk (checkpointing). The pup framework can be extended to provide services to any operation that requires a traversal of the object state (typically a traversal over the objects data members).

To migrate concurrent objects such as chare array elements the following steps need to be done,

1. the state of the object must be 'packed' into a memory buffer
2. the memory occupied by the object should be released on the processor where the object resided
3. the object state should be transported to the new processor where the object is to be migrated
4. the object should be recreated at the new location.

The state of sequential objects associated with the Chare Arrays is subsumed by the state of the concurrent objects since, in a Charm++ program, every sequential object is a member of or is pointed to by a member of a concurrent object. Chare Arrays may contain dynamically allocated data the size of which varies at runtime. All of this data needs to packed at the time of migration from the source processor and unpacked at the destination processor.

To checkpoint a Charm++ program, we need to save the state of all the concurrent objects in the system to disk. We chose to view checkpointing as a variant of migration.

When a checkpoint is made, the Charm++ objects are seen as 'migrating' to disk, and upon restart they return to their respective processors. To migrate an object, its data needs to be 'packed', i.e. serialized either into a memory buffer or to disk, and then 'unpacked' into memory.

Migration for objects can be handled in several ways. A possible approach is to require each class to implement **pack** and **unpack** methods. If an object is required to migrate to another processor while the program is executing, the method **pack** is invoked on the object. The pack method allocates a memory buffer large enough to hold all the objectś data and then proceeds to serialize the objects data into the memory buffer. The memory buffer is then inserted into a message and sent to the processor where the object is to be migrated. When this message containing the object state is received by the new processor, a new instance of the class is created by calling a special *migration constructor*. The migration constructor's task is to simply create an uninitialized instance of the class. The **unpack** method is invoked with a pointer to the migration message. The unpack method proceeds to stuff the new object with data from the old one. At the end of the unpack method, migration is complete.

The problem with this approach is that most of the functionality in the pack and unpack is similar in nature, i.e., in the pack function, the data is copied to a serial buffer in a particular order and in unpack the data is copied from the serial buffer in the same order as it was packed. Thus there is code duplication as both the methods share a common skeleton, with only the actual operation that the methods perform on the data members being different. The pup library has been designed with the intent of preventing this code dupplication. The programmer of a particular class only needs to implement a single method, called *pup*. The *pup* method takes a single parameter, which is an instance of a *packer/unpacker* or *pupper*. The nature of puppers shall be dealt with subsequently. The role of this method is to perform a traversal of the object state. The actual operations that need to be performed on the data members are executed by the pupper.

The pup library contains the following important classes:

- `class PUP::er` - This class is the abstract superclass of all the other classes in the system. The *pup* method of a particular class takes a reference to a *PUP::er* as parameter. This class has methods for dealing with all the basic C++ data types. All these methods are expressed in terms of a generic pure virtual method. Subclasses only need to provide the generic method.

- `class PUP::packer` - The abstract superclass of all classes that 'pack' objects. It is not clear here what packing means, but it may be considered as any operation that performs a non-destructive transformation on the objects state, i.e. the 'packing' operates on the data that constitutes the object state and creates a different representation of that state. The object does not change as a result of this operation. This class implements additional methods, `isPacking` and `isUnpacking`, that may be used to query the class to determine its mode of operation

- `class PUP::unpacker` - The abstract superclass of all classes that 'unpack' objects. Unpacking is the opposite of packing as described in the previous item. An unpacking operation works with a 'raw' (uninitialized) object and some representation of the object state. The process of unpacking involves a traversal of the object state, at each step of the traversal, part of the object state is 'converted' from the given representation into a piece of memory holding the right bit pattern. When the unpacking is complete, the entire object state has been recovered.

- `class PUP::sizer` - This is a subclass of the `PUP::er` class. Its function is to determine the size (in bytes), of the object that it operates on.

- `class PUP::toMem` - This is a subclass of the `PUP::packer` class. The role of this class is to pack the object it operates on into a preallocated contiguous memory buffer. The most general way to pack an object into a memory buffer is to invoke `pup` on the object using an instance of `PUP::sizer` to determine the size of the object, then a buffer of the required size is allocated and `pup` is invoked again with an instance of `PUP::toMem` that has been initialized with the allocated buffer.

- `class PUP::fromMem` - This is a subclass of the `PUP::unpacker` class. The role of this class is to unpack the state of the object it operates on from a given contiguous memory buffer.

- `class PUP::toDisk` - This is a subclass of the `PUP::packer` class. The role of this class is to save the state of the object it operates on into a disk file. To serialize an object to disk, `pup` is invoked on the object with an instance of `PUP::toDisk` that has been initialized with a file pointer.

- `class PUP::fromDisk` - This is a subclass of the `PUP::unpacker` class. The role of this class is to unpack the state of the object it operates on from a given disk file.

Figure 3.1 shows the shows a class declaration that includes a `pup` method:

Figure 3.1: A simple class declaration showing the `pup` method

```
class foo {
 private:
  bool isBar;
  int x;
  char y;
  unsigned long z;
  float q[3];
 public:
  void pup(PUP::er &p) {
    p(isBar);
    p(x);p(y);p(z);
    p(q,3);
  }
};
```

The routine in Figure 3.2 presents an example of how an instance of the *foo* class may be packed and unpacked from a memory buffer.

Figure 3.2: Packing and unpacking a *foo* object

```
int main()
{
  //Build a foo
  foo f;
  f.isBar=false;
  f.x=102;f.y='y';f.z=1234509999;
  f.q[0]=(float)1.2;f.q[1]=(float)2.3;f.q[2]=(float)3.4;

  //Collapse f into a memory buffer
  //Allocate a buffer for the foo object
  PUP::sizer s;
  f.pup(s);
  void *buf=(void *)malloc( s.size() );
  //Pack f into preallocated buffer
 {PUP::toMem m(buf);f.pup(m);}

  //Unpack the foo object
  foo f2;
  {PUP::fromMem m(buf);f2.pup(m);}
}
```

The following more complex example shows how an instance of the *bar* class may be serialized to memory and then recovered from it. The *bar* class has an instance variable of type *foo*. To pack/unpack or checkpoint/recover an object of type bar we must apply the same operation to the instance variable `foo f`. This is accomplished by having the `pup` method of the bar class invoke `pup` on the member of type foo with the *pupper* passed to it. Figure 3.3 shows the declaration of the *bar* class, including the `pup` method.

Figure 3.3: Declaration of the *bar* class showing the `pup` method.

```
class bar {
 public:
  foo f;
  int nArr;//Length of array below
  double *arr;//Heap-allocated array

  bar() {}
  bar(int len) {nArr=len;arr=new double[nArr];}

  void pup(PUP::er &p) {
    f.pup(p);
    p(nArr);
    if (p.isUnpacking())
      arr=new double[nArr];
    p(arr,nArr);
  }
};
```

Chapter 4

Load Balancing

Accessing the power of parallel computation demands efficient load distribution across the processors of the parallel machine. Applications where parallel processing has succeeded impressively are those where the problem can be decomposed into convenient number of equal sized partitions. Applications like AMR, which are irregular or dynamic in their structure, prove more difficult to implement with good parallel efficiency. [4] shows that treating a program as a collection of communicating objects, measuring the execution time consumed by those objects at run time, and achieving good load balance by automatically moving those objects from processor to processor allows the implementation of efficient load balancers that work with a variety of applications. This chapter discusses the load balancing strategies that are implemented in the Charm++ distribution and which can be used in the applications constructed using the AMR library. The text for this chapter has been composed from [4] and [24]. [4] details how to implement a new load balancing strategy for the load balancing framework in Charm++.

4.1 Load Balancing Strategies

The fundamental tradeoff in parallel load balancing is the balance between execution time and load balancing time. Load balancing algorithms may be able to improve the performance, but if the cost of load balancing consumes more time than is gained by better load distribution, the load balancer is not worthwhile. Typical parallel applications have a long execution time, thus making it worthwhile to do the load balancing. Various strategies implemented in the Charm++ distribution can be categorised into the following:

- Centralized Load Balancing
- Neighborhood Load Balancing

4.1.1 Centralized Load Balancing

Centralized load balancing refers to the strategy of having each executing thread synchronize at specific points in the application code. Centralized load balancing is used when the load balancing strategy requires global information about the state of the system. Various strategies for load balancing that fall under this scheme are

- **RandomLB** RandomLB performs random load balancing. This strategy merely picks a random destination for each object. This can result in reasonable load balance if communication overhead is small and there are a large number of objects. Since each object is assigned to a processor by selecting random destination, RandomLB requires a single pass through the object list, and the computational complexity of RndomLB is O(N).

- **GreedyLB** In GreedyLB, the objects in the object-communication graph are redistributed without regard to communication or current processor assignment. A processor heap is built so that the processor with the least assigned load is at the top of the heap. Initially, 0 objects are assigned to every processor (so every processor in the heap has no load) and the processor at the top is arbitrary. An object heap is also built based on the object-communication graph, and the object taking the most time is placed at the top of the heap. For N objects this strategy has O(Nlog N) complexity, but can result in the migration of the majority of the objects.

- **RefineLB** RefineLB is a strategy which improves the load balance incrementally by adjusting the object distribution. In this algorithm the load is adjusted on the basis of the computed average load and distributing the load from the heavily loaded processors to the lightly loaded processors. Use of this strategy results in the migration of very few objects, as objects are relocated to relieve the overloaded processor.

- **RandomRefineLB and GreedyRefineLB** RandomRefineLB and GreedyRefineLB are two strategies that apply the refinement algorithm after applying the random and greedy strategies respectively. In most cases, we expect refinement to only produce small improvements, but for cases where the base strategy has resulted in a few badly overloaded processors, the refinement step may provide dramatic performance improvement.

- **RecBisectBfLB** RecBisectBfLB is a load balancer which uses the object-communication graph to recursively partition the objects until there is one partition for each processor. Each partition is assigned to one of the processors. This method provides improved communication performance, since communicating objects are likely to be assigned to the same processor. This strategy may do a poor job at times, since each partition does not have the freedom to fine tune its load by selecting some unconnected objects.

4.1.2 Neighborhood Load Balancing: NeighborLB

Another approach to load balancing is to reduce the message traffic and synchronization overhead by performing neighborhood load balancing. This is implemented in the NeighborLB strategy. In this strategy, each processor sends summary statistics about its load to a neigborhood of processors. When each processor has recieved load information from all its neighbors, it compares its load to the average of the neighborhood. If it is below the average then it tells its neighborhood that no objects need to be migrated from its processor. If it is overloaded, it repeatedly selects the largest of its objects which can be passed to the lightest loaded neighbor without overloading that neighbor, until the processor load drops below the neighborhood average, or none of the processorś objects will fit on any of the neighbors without overloading them. Each neighbor is informed which objects will be arriving, and then the load coordinator is asked to move those objects away. When all elements have arrived, NeighborLB strategy asks the load coordinator to resume the synchronised objects, and execution continues. NeighborLB at this time uses a neighborhood of four processors but this is easily modifiable.

4.2 Load Balancing Strategies API

In order to use a load balancing strategy, there are three things that need to be done:

- extern the load balncing strategy module in .ci file.
- include the appropriate header file (based on the load balancing strategy) in the .h file
- make a call to start the load balancer from main::main (constructor for the mainchare) in .C file

For the RefineLB strategy, for e.g., the following needs to be done:

```
In .ci file
extern module RefineLB;


In .h file
#include<RefineLB.h>


In .C file
main::main() {
...
```

Table 4.1: API for various load balancing strategies

Strategy	**modname**	**headerFile**	**CreateFunc**
RandomLB	RandomLB	RandomLB.h	CreateRandomLB()
GreedyLB	GreedyLB	GreedyLB.h	CreateGreedyLB()
RefineLB	RefineLB	RefineLB.h	CreateRefineLB()
RandomRefineLB	RandomRefineLB	RandomRefineLB.h	CreateRandomRefineLB()
GreedyRefineLB	GreedyRefineLB	GreedyRefineLB.h	CreateGreedyRefineLB()
RecBisectBfLB	RecBisectBfLB	RecBisectBfLB.h	CreateRecBisectBfLB()
NeighborLB	NeighborLB	NeighborLB.h	CreateNeighborLB()

```
CreateRefineLB();

...

}
```

In general, the three things that need to be done for any load balancing strategy can be represented as:

In .ci

extern module modname;

In .h

#include<headerFile>

In .C

main::main{ CreateFunc Call }

Table 4.1 shows the corresponding API for all strategies discussed in this chapter.

Chapter 5

Adaptive Mesh Refinement Library

The AMR library provides the user the ability to implement an application using the adaptive mesh refinement technique in parallel using *Charm++*. This library can be used to implement AMR applications in 1D, 2D or 3D. It takes care of the implementation details of binary, quad or oct trees, depending on whether you are using 1D, 2D or 3D AMR respectively. It also takes care of neighbor communications and refinement/coarsening. The library allows the user to do dynamic load balancing to increase performance. This chapter describes the design of the AMR library and discusses the performance of the library for a sample application.

Figure 5.1: Mapping of grid to the quad tree in 2D AMR

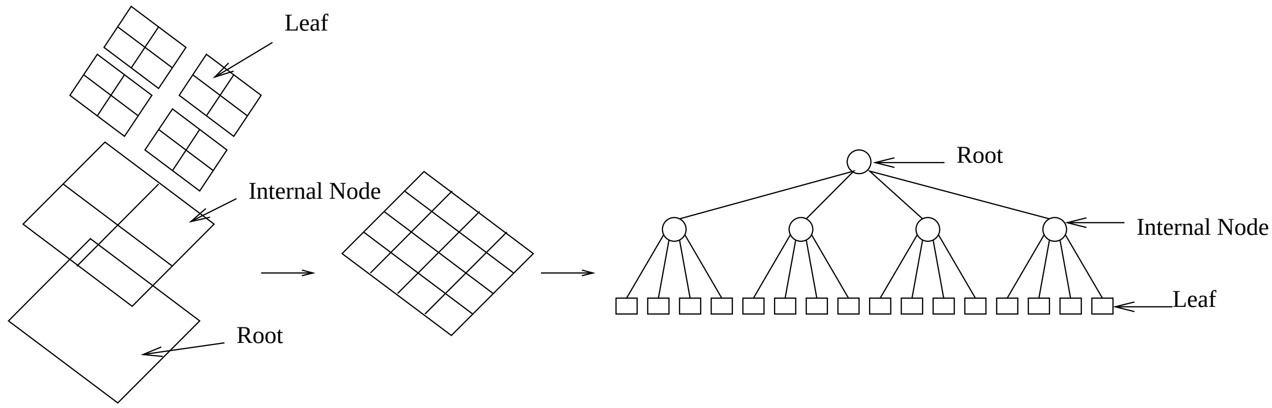


Figure 5.2: Typical algorithm for application with nearest neighbor communication pattern

```
steps = totalSteps
do while steps > 0
  //send and receive boundaries
  communicateWithNbors()
  doComputation()
loop
```

5.1 Introduction

In an AMR application, the basic data structure is a tree. Each node in the tree is called a 'Cell'. The Cells at the bottom of the tree are called 'leaves'. Only the leaves take part in the computation. Each leaf typically has a part of the grid. All the children of a Cell combine to form a coarser grid. The root of the tree represents the coarsest level of the grid. Figure 5.1 shows the mapping of a grid in a tree. Each leaf does the computation on its subgrid and then communicates the boundaries to its neighbors. The communication pattern is very similar to the nearest neighbor communication pattern. Figure 5.2 shows the general algorithm for an application with nearest neighbor communication pattern. The AMR algorithm (Figure 5.3) is very similar to this with the addition of checking of refinement criterion to adaptively refine the tree.

For the sequential version of AMR, if the application is being designed from scratch, code needs to implemented for the following:

1. Tree with dynamic addition and deletion (binary tree, quad tree or oct tree for 1D, 2D or 3D AMR respectively)

2. Strategy for determining the neighbors of the leaf

3. Communication of the boundaries between neighbors

4. Application specific data structures for the Cells

5. Initialization of the data in the Cells at the beginning

Figure 5.3: Basic algorithm being executed in parallel

```
steps = totalSteps
refineStep = refineInterval
do while steps > 0
  communicateWithNbors()
  doComputation()
  if step = refineStep
    synchronise()
    checkForRefinement()
    if Invariant violated
        ask neighbors to refine
    if refined or AutoRefined
        changeChildrenToLeaves
        exit while loop
    else
        refineStep += refineInterval
        step += 1
loop
```

6. Body of computation performed by the leaf in each iteration

7. Criteria for deciding if a leaf needs to be refined or coarsened.

8. Methods for combining or extrapolating data recieved from neighbors at different levels of refinement.

9. Methods for dividing the existing data in a Cell for children during refinement.

If the application is being written in parallel, then the first three steps, mentioned in the above list, become more complex as they have to be done in parallel. Additional issues like load balancing, further complicate the application development. The last four steps in the list are application specific and need to be done differently for each application. The AMR library takes care of the first three steps and the load balancing. Thus the user has to implement only the application specific code which greatly simplifies the application development.

Figure 5.4: Creating the library from the mainchare

```
/*In .ci file */
mainmodule pgm {
 extern module amr;
 initcall void AmrUserDataJacobiInit(void);
 mainchare main
 {
   entry main();
 };
};
/*In your .h file*/
#include "amr.h"
class main : public Chare {
 public:
main();
};
/*In .C file */
#include "pgm.decl.h"
...
main::main(CkArgMsg* args) {
    StartUpMsg *msg;
    msg = new StartUpMsg;
    msg->synchInterval = 200;
    msg->depth = 2;
    msg->dimension = 3;
    msg-> totalIterations = 500;
    CProxy_AmrCoordinator::ckNew(msg,0);
}
...
#include "pgm.def.h"
```

5.2 Library's User Interface

The Library Interface was designed such that the user's code for the parallel application should be similar to the sequential code. The user writes code from the point of view of a single, uniformly refined, grid and provides the methods explained in section 5.2.2. This section describes the interface of the library briefly. For a detailed explanation of the interface refer to the AMR manual[25].

5.2.1 Creation of Library

The library is created in the constructor for the class main, which inherits from a class called *Chare* as shown in Figure 5.4. The library is created by calling *ckNew* method of the class *CProxy_AmrCoordinator*.

CProxy_AmrCoordinator::ckNew(msg,0);

5.2.2 User Data Class

The library makes no assumption about the users data structure. The user should implement their data structure in a class which inherits from the class AmrUserData. This class implements the data structure for each leaf. The library requires this class to implement the following methods:

`void doComputation(void)`

//Computation

This method is used to implement the computation to be done by each leaf. For each iteration, this method is invoked by the library after communicating (send and receive) data with all the neighbors.

`void** getNborMsgArray(int* sizeArray)`

//Neighbor Communication: Get the data to be communicated

This method is called to get the data to be communicated to the user. It is expected to return an array of data arrays (allocated by the user) to be communicated to the neighbors. The size of the array of data arrays depends on the dimensionality of AMR that is being used as shown in Table 5.1. Each data array is sent to a different neighbor, hence the library expects the data arrays to be in the order specified by Table 5.2. The argument to the function is an array containing the sizes of the data arrays. The user must specify the size of each data array in bytes in the order specified in Table 5.2.

Table 5.1: Size of Array Returned by getNborMsgArray

AMR Type	Return ArraySize Required
1D	2
2D	4
3D	8

Table 5.2: Message Ordering in the array returned by getNborMsgArray

ArrayIndex	For Which Neighbor	AMR Type
0	+X Nbor(Right)	1D,2D,3D
1	-X Nbor(Left)	1D,2D,3D
2	+Y Nbor	2D,3D
3	-Y Nbor	2D,3D
4	+Z Nbor	3D
5	-Z Nbor	3D

```
void store(void* data , int dataSize, int neighborSide)
```

//Store the data received from the Neighbor

This method is called by the library to give the user the data received from the neighbor. The user should copy the data into a local data structure. The data should not be stored directly by storing the pointer. The arguments of this method have the following interpretation

- *data:* a pointer to the data received from the neighbor.
- *dataSize:* the size of the data received in bytes.

- *neighborSide:* which neighbor sent the data. Table 5.3 explains what the various values mean.

Table 5.3: Various neighborSide values passed

neighborSide	From Which Neighbor	AMR Type
0	-X Nbor(Left)	1D,2D,3D
1	+X Nbor(Right)	1D,2D,3D
2	-Y Nbor	2D,3D
3	+Y Nbor	2D,3D
4	-Z Nbor	3D
5	+Z Nbor	3D

```
void ** fragmentNborData(void* data, int *sizePtr)
```

//fragment neighbor data received to be sent to the finer neighbor

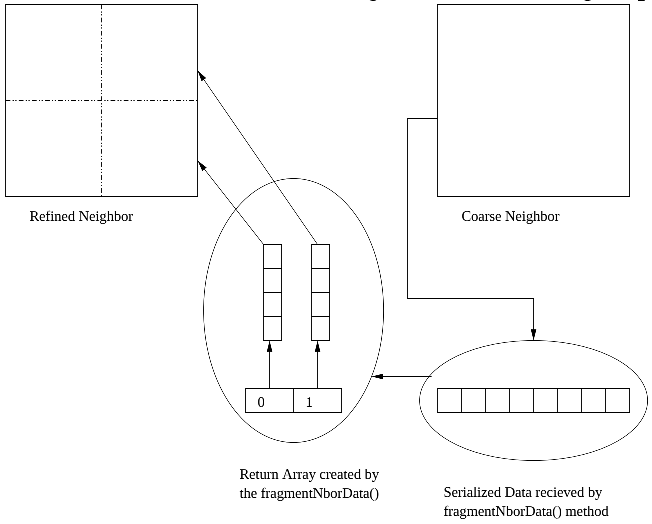
This method is called by the library only when adaptive refinement is being used. If a message is to be communicated from a coarser level to a finer level the message sent needs to be broken up in the case of 2D and 3D AMR. The pointer to the array of data and a pointer to its size in bytes is passed to the method to fragment it into 2 parts for 2D AMR and 4 parts for 3D AMR. The user's code is supposed to fragment the data received into an array of fragmented data arrays and update the size in bytes pointed to by the sizeptr to the new size in bytes of the fragment (Assumption all fragments are of the same size). Figure 5.5 shows how the fragmentNborData should work. If there is a need to extrapolate these fragmented messagesit should be done here, or in the store method (where these messages will be received).

```
void combineAndStore(void **dataArray, int dataSize, int neighborSide)
```

//combine the data received from the finer neighbors and store the data

This method is used for receiving the message from fine neighbor. For 2D AMR dataArray has 2 messages whereas in 3D AMR dataArray has 4 messages. The message received may need to be interpolated before being stored (library does not provide any methods to do that

Figure 5.5: Message Splitting for 2D AMR



as it may be application specific). The function is similar to store in all other respects.

`bool refineCriterion(void)`

//refinement criterion to be used at the refinement stage

The library calls this method on the user data class (leaves in the tree) periodically (frequency specified by the user during the creation of the library). This method determines if refinement is needed on the basis of local data in the leaf.

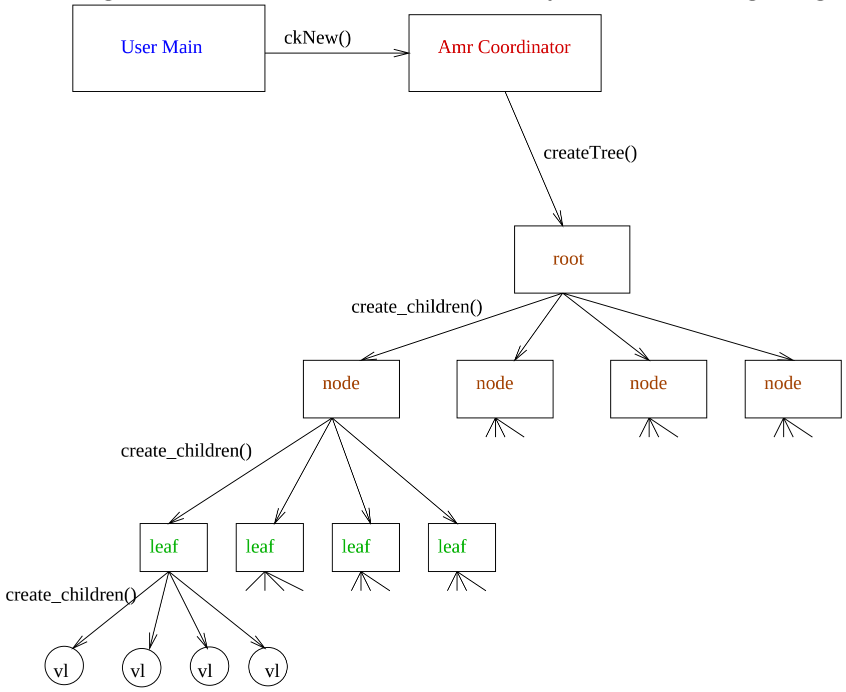
`void **fragmentForRefine(int *sizePtr)`

//fragment the leaf and divide the data for the children

This method is called if the leaf needs to be refined which is determined either on the basis of refinement criterion or to preserve the invariant in the tree (see section 5.3.2). In this method the user is required to split the data into 2, 4 or 8 parts for 1D, 2D or 3D AMR respectively. Each fragment is sent to one of the children of the leaf being refined.

Detailed explanation of these methods is given in [25]. Besides these methods the user might want to implement the interpolation/extrapolation, boundary constraint and utility methods to be used by the class. The library does not make any assumption about the naming of these methods and it is the responsibility of the user to call these methods when appropriate.

Figure 5.6: Order of creation of objects at the begining of execution for 2D AMR



5.2.3 Load Balancing

In order to use the load balancing strategy the user must call the corresponding function to create the strategy in the constructor for mainchare (as explained in Section 4.1)

5.2.4 Flow of Execution for the Library

Figure 5.6 shows the order of events in the beginning of the execution of an AMR application. User's main instantiates the *AmrCoordinator*, which in turn creates the root of the tree. The root then creates the children until the desired level (specified by the user). After creation is done, each leaf starts communication and computation. The leaves communicate with their neighbors and do computation until the time comes to check if refinement is needed. The user specifies at the beginning of the program when the library should check for refinement. If refinement is needed, children are added for the leaf and the library makes sure that the neighbors maintain the invariant (explained in Section 5.3.2). These steps are repeated until all the iterations are complete. Figure 5.3 shows the steps being followed in parallel by each leaf.

5.3 Library Design

This section discusses the design of the AMR Library. It is subdivided into three sections for more lucid explanation of the issues.

5.3.1 Hierarchical Indexing of Tree

The indexing of the array representing the tree (binary, quad or oct) is an important issue in the design of this parallel library.

A naive way of indexing the array would be to index the elements in the tree sequentially. In this naive approach, a central object keeps track of the structure of the tree. Every leaf asks a central object who its neighbors are in each iteration. It is trivial to see that the central object would soon become a bottleneck because it has to deal with the insertions in the tree due to refinement and respond to the neighbor queries from the leaves. Hence this scheme will fail to scale well with increase in the size of the tree.

Another approach could be each leaf caches the neighbor indices and asks for the updates when refine/coarsening takes place. Though this modified scheme is better than the previous one, it still has a central object as the bottleneck because the central object still has to handle the assignment of a unique index for the new children and calculate the new neighbors.

From the above two examples, we can see that a good distributed indexing scheme will have the following features

- The scheme should be able to determine a unique index for the children without complete knowledge of the tree structure.
- The indexing scheme should allow a leaf to determine its neighbors locally.

Warren et al. give a hierarchical indexing scheme for oct trees in [31]. In their scheme, the children derive their index from their parent which ensures its uniquenes. The AMR library uses a similar scheme for indexing the tree, with some modifications, for ease of implementation and local detrmination of neighbors. Such an indexing scheme was also used in parallelization of the search trees in [17] at the Parallel Programming Laboratory, University of Illinois at Urbana-Champaign.

In the indexing scheme used in this library, the index of a node is represented by a bitvector in each dimension and the total number of bits used (sum of bits used in each dimension), i.e.

```
((x):n) in case of 1D,
((x,y):n) in case of 2D and
((x,y,z):n) for 3D and so on.
where x is a bitvector in x dimension.
      y is a bitvector in y dimension.
      z is a bitvector in z dimension.
      n is total number of bits used.
```

The index of the root is a 0 bitvector in each dimension and the number of bits used is also 0, i.e., ((0):0) in 1D, ((0,0):0) in 2D, and ((0,0,0):0) in 3D. The index of a child is determined on the basis of the index of the parent. For simplicity take the case of a 2D index.

```
Let the parent be
((x,y):n)
Thus the indices for its four children will be
((2x,2y):n+d)
((2x+1,2y):n+d)
((2x,2y+1):n+d)
```

```
((2x+1,2y+1):n+d)
where x,y are bitvectors in x and y dimension respectively
      n is total number of bits for parent
      d is the dimension of AMR (in this case 2)
```

Lemma 5.1 *Proof of Uniqueness of the Indices*

It can be trivially seen, from the scheme to obtain the indices of children above, that all the children of the same parent will have different indices. Thus, based on the above assertion the proof can be divided into 2 parts:

- *unique indices for nodes at different levels in a tree*
- *unique indices for nodes at the same level but having a different parent*

The nodes at different levels of the tree will all have different indices as they will have different number of bits (n) representing the bitvectors. Since the number of bits is part of the index it can be stated that the index for nodes at different levels of tree will never be the same.

For the nodes at the same level but from different parent we will prove by contradiction that they will be unique. Assume that $((x, y) : n)$ *and* $((x', y') : n)$ *are indices of children of different parents but they are equivalent, i.e.*

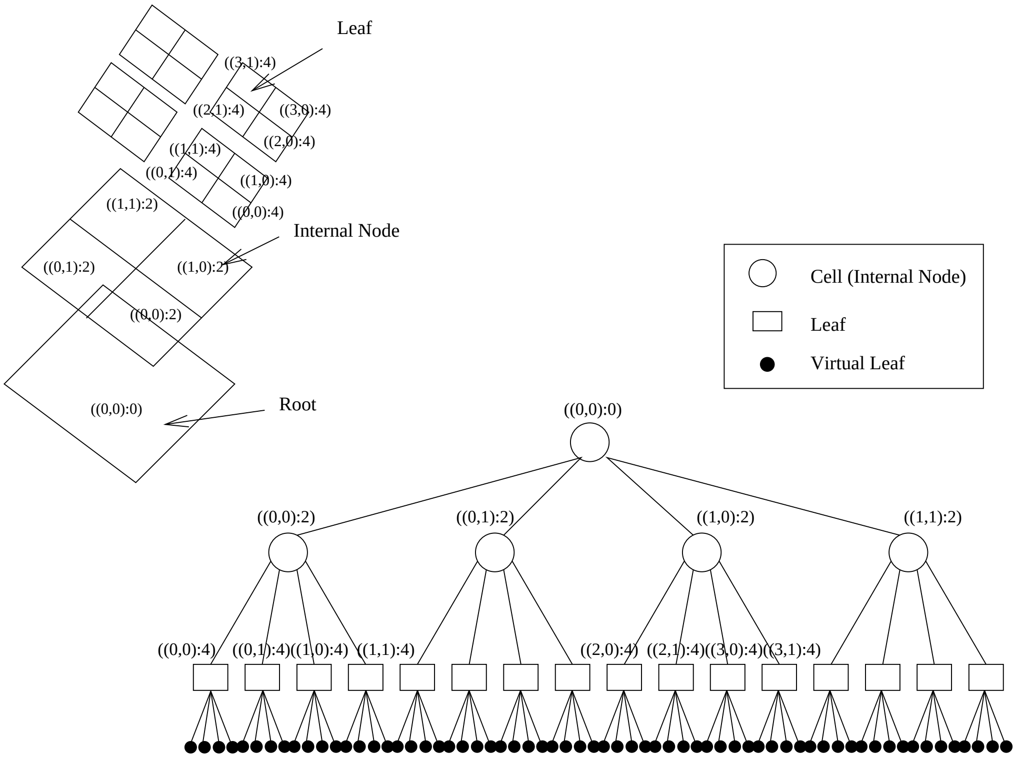
$$x = x' \tag{5.1}$$

$$y = y' \tag{5.2}$$

$$n = n \tag{5.3}$$

If we right shift the bitvectors in both the x and y dimension by 1 bit and subtract 2 from the number of bits (n) we will get the index of the parent. So the parent for $((x, y) : n)$ *will be* $((RShift(x, 1), RShift(y, 1)) : n - 2)$*. Similarly, for* $((x', y') : n)$ *the parent will be* $((RShift(x', 1), RShift(y', 1)) : n - 2)$*. Now we can see that from Equation 5.1, 5.2 and*

Figure 5.7: Indexing in 2D AMR (quad Tree)



5.3 that both the bitvectors for the parents will be the same which is contrary to the hypothesis that they have different parents. Thus we have proved by contradiction that all the nodes in the same level of the tree but from a different parent have different indices.

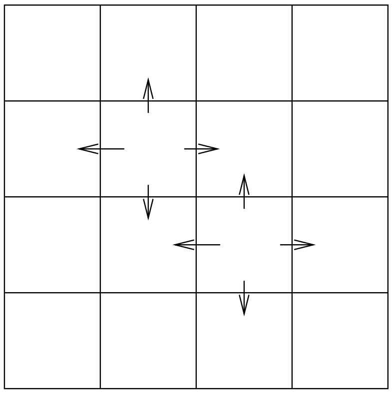
So we can conclusively say that all the indices in the tree will be different.

Figure 5.7 shows indexing in the case of 2D AMR (i.e. for a quad tree). Based on the indexing scheme explained above, each leaf can determine unique indices for its children locally.

5.3.2 Neighbor Communication

In AMR applications, the communication pattern is like nearest neighbor, i.e. each leaf talks to its neighboring leaves. Each leaf can have a maximum of 2, 4 or 8 neighbors for 1D, 2D or 3D AMR respectively (diagonal neighbors are ignored in this discussion). Figure 5.8 shows the nearest neighbors for a leaf in consideration for 2D AMR. The indexing scheme makes

Figure 5.8: Nearest Neighbors for the leaf



it easy to determine the indices of neighbors locally. In order to simplify the discussion we define two kinds of neighbors in a dimension:

- -ve neighbor: This neighbor is on the lower coordinate side of the current node (Equivalent to left in the x dimension).

- +ve neighbor: This neighbor is on the higher coordinate side of the current node (Equivalent to right in the x dimension).

In order to find neighbors in d^{th} dimension we can use the following formulas:

-ve neighbor : b - 1 | $b > 0$

+ve neighbor : b + 1 | $0 \leq b < 2^n$

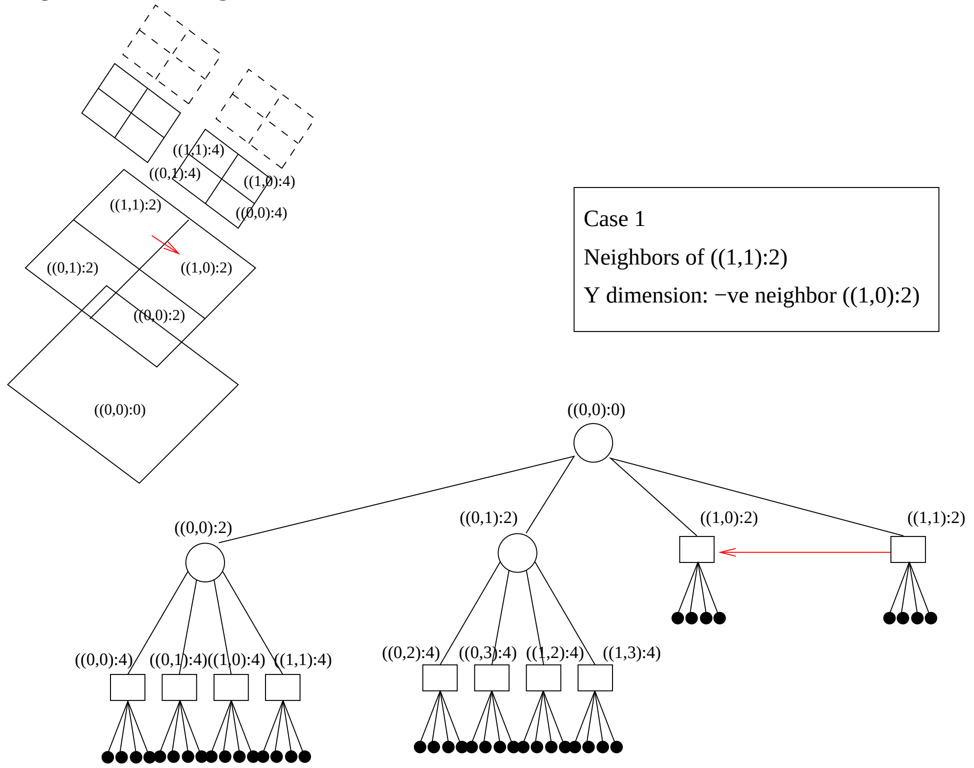
where b is the bitvector in d^{th} dimension; and

n is the number of bits used in b

If a leaf has a neighbor at the same level as its own then it is very simple to determine the neighbors address with the above formula. Figure 5.9 shows an example where a node with index ((1,1):2) sends a message to its -ve neighbor in y dimension with index ((1,0):2). The formula given above to calculate the neighbors works well if all the leaves are at the same level. Taking another example, the neighbors for ((1,1):4) according to the above formula will be:

X dimension: -ve neighbor ((0:1):4)

Figure 5.9: Neighbor Communication:Sender and Receiver at the same level in a quad tree



X dimension: +ve neighbor ((2:1):4)

Y dimension: -ve neighbor ((1:0):4)

Y dimension: +ve neighbor ((1:2):4)

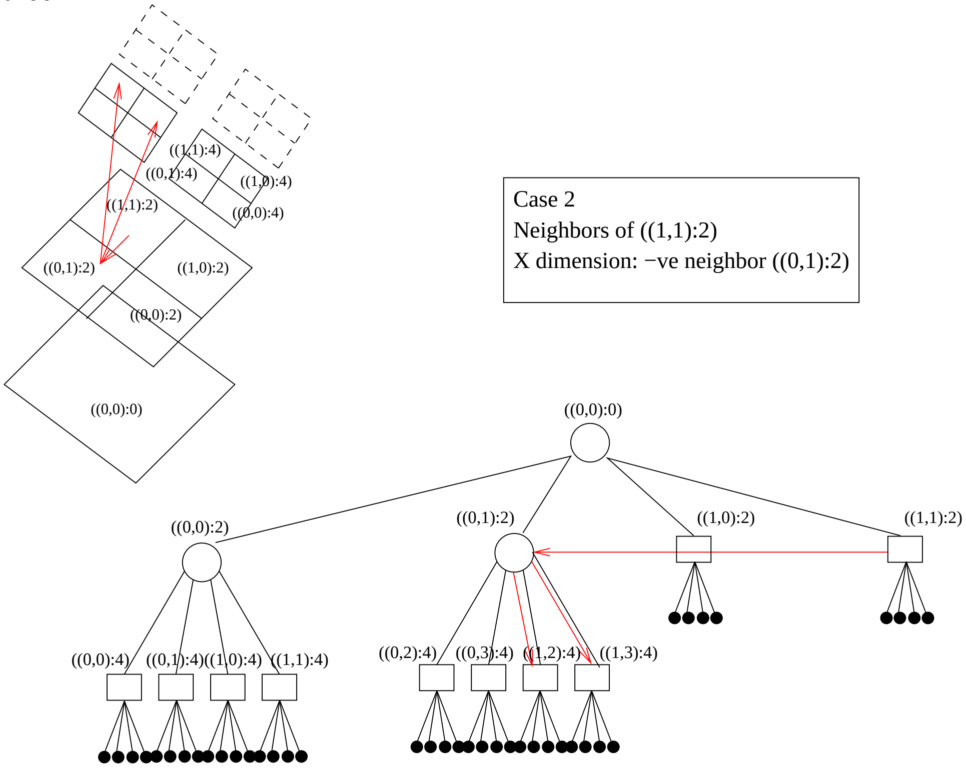
From Figure 5.9 we can verify that the calculated neighbors according to the formula are correct. Hence we can say that the formula for calculating neighbors works at the same level of refinement, for not only neighbors with a common parent but also for the neighbors with different parents.

If the leaves are at different levels then the formula above will not work directly. This situation can be divided into two cases:

Sender is coarser than the receiver

If the sender is coarser (i.e. up in the tree) by 1 level, then the receiver's parent will receive the message. The parent will have to split the message before forwarding it to its children as shown in Figure 5.10. In Figure 5.10 the sender ((1,1):2) sends a message to its -ve neighbor

Figure 5.10: Neighbor Communication:Sender at a coarser level than the Receiver in a quad tree



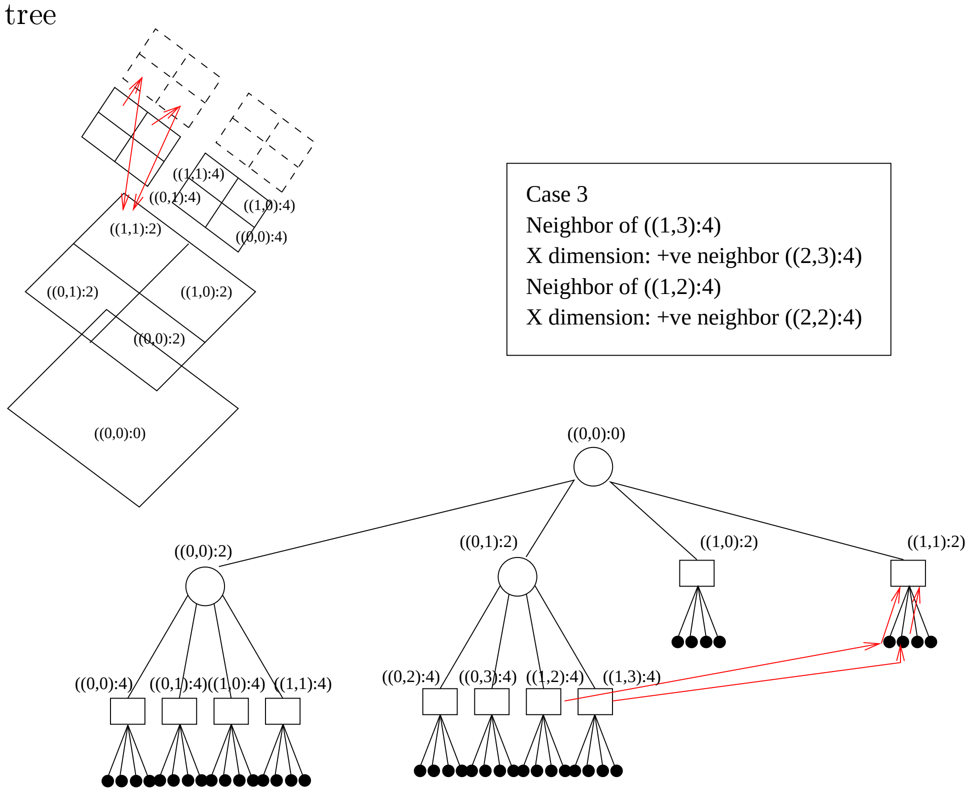
in the x dimension, ((0,1):2), which is more refined. When ((0,1):2) recieves the message it splits the message into two parts, using a user supplied interpolation function, and forwards the split messages to the appropriate children.

Sender is finer than the receiver

If the sender is finer (i.e. down in the tree) by 1 level, then the receiver's children will receive the message. Since the receiver is always a leaf this scheme will fail as the leaf does not have children. To circumvent this problem, we introduce a level of virtual leaves below the real leaves, i.e. each leaf will have virtual leaves as their children. These virtual leaves, implemeneted as regular parallel objects, receive the message from the finer sender and forward it to their parent (leaf). Figure 5.11 shows the case if sender is finer than the receiver.

From the above discussion we can see that the indexing scheme will only work if the difference in refinement levels of the neighbors is at maximum 1. So we maintain the following

Figure 5.11: Neighbor Communication: Sender at a finer level than the Receiver in a quad tree



invariant in the tree through out the execution of the program:

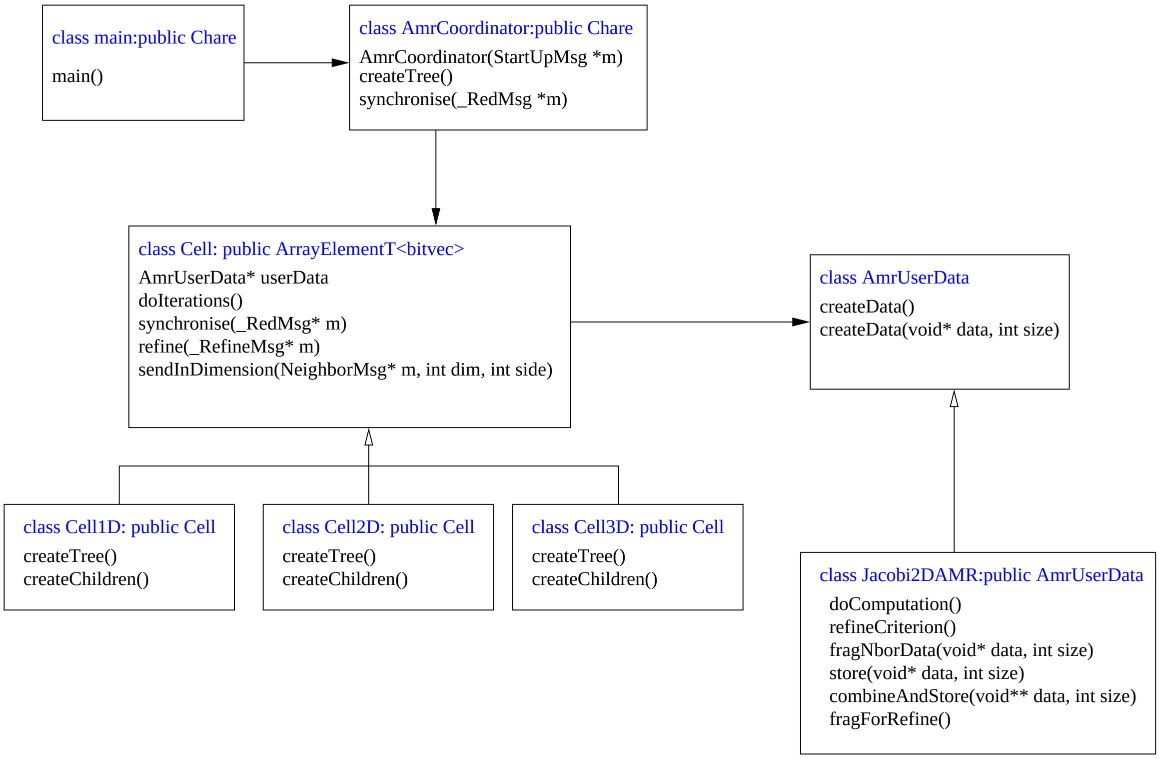
Invariant At all times, the maximum difference between the levels of refinement of neighbors is 1.

We have to make sure that this invariant is not violated at any time in the tree. Whenever there is a need for refinement in the tree, we also check if the neighbors need to refine to preserve the invariant.

5.3.3 Class Design

Figure 5.12 shows the class diagram for the AMR library. This diagram shows that the user writes a *main* class which inherits from the class *Chare*. This class creates an instance of the library by instantiating *AmrCoordinator*. *AmrCoordinator* is a class which inherits from *Chare* class, and its instance can only be created on processor 0. This constraint is imposed

Figure 5.12: Class Diagram for the AMR Library



by the current implementation of Charm++ which allows chare arrays to be created only from processor 0. *AmrCoordinator* creates the root of the tree, and keeps a proxy to it in the cache for later interaction. The nodes in the tree are instances of class *Cell1D*, *Cell2D* and *Cell3D* for 1D, 2D and 3D AMR respectively. *Cell* is the abstract super class for *Cell1D*, *Cell2D* and *Cell3D*. *Cell* implements all of the common functionality which is independent of the dimension of AMR being used. *Cell* makes use of template methods and factory methods[9] to implement some of this functionality. At this time this library supports 1D, 2D and 3D AMR but the design is general enough to support higher dimensions. Each leaf has an instance of the user's data class (e.g Jacobi2DAMR is such a class in Figure 5.12). User's data class should inherit from *AmrUserData* and should implement the interface specified by it. No assumption is made about the data structure used in user's data class, which makes this library different from most of the other implementations of the AMR libraries.

5.4 Preliminary Performance Measurement

The results presented in this section are preliminary results without any performance tuning. Performance tuning for this framework is an area not covered in this thesis and is a ripe topic for subsequent research.

5.4.1 Benchmark Application

We developed a fluid dynamics application[1] to run as benchmark for the library. In this application the fluid area is divided into regular rectangular grid and we keep track of the fluid's pressure and velocity at every grid point. We have fixed time step for the grid and in each time step we recompute the pressure and velocities at each grid point. The new pressure and velocity for a grid point is computed based on the previous time step values of itself and the immediate neighboring grid points. The boundaries of the grid are like reflecting walls, i.e. they have zero velocity and pressure is the same as the neighbor. As the initial condition, we have a bar of high pressure in one corner of the grid. As the simulation progresses, this bar of pressure shoots out on both sides and bounces of the walls until it diffuses out to an even shade. The grid is refined in the areas where there is a high pressure gradient.

5.4.2 Results

We ran the benchmark on a 6 node (quad Xeon processors) cluster. The benchmark was run with no load balancing, heapCentralLB and refineLB strategies for different number of processors as shown in the graphs. The time shown in the graphs is the wall clock time taken for the complete run and includes all the overhead incurred.

By looking at the graphs (Figure 5.13 and Figure 5.14) we can see that if the load balancing strategy does not take communication into account, then the overhead is more

[1]The serial version of this application was developed by Orion Lawlor

Figure 5.13: Performance Graph without load balancing

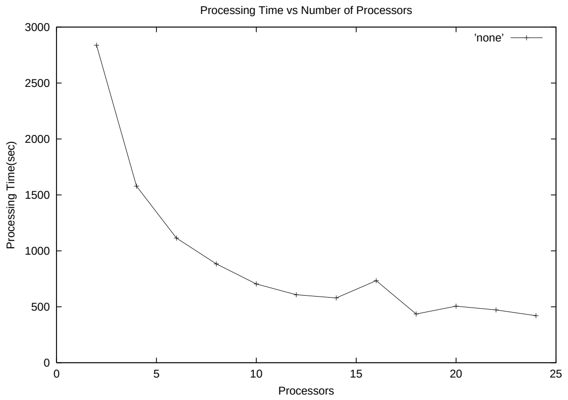


Figure 5.14: Performance Graph with heapCentral load balancing

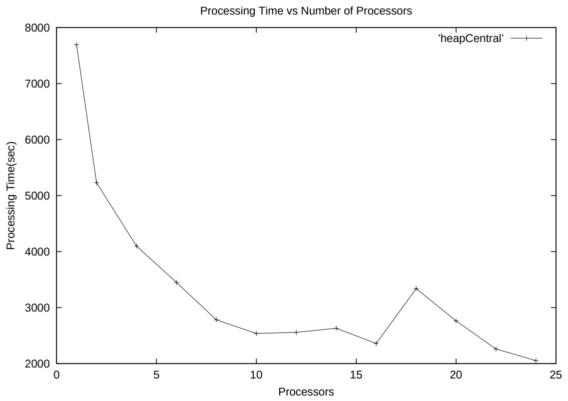
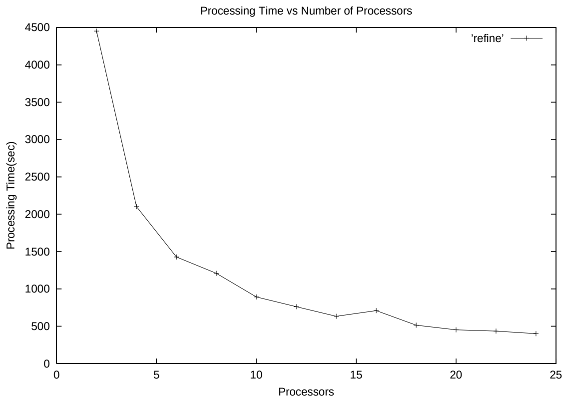


Figure 5.15: Performance Graph with refine load balancing



than its usefulness. Also by looking at the graphs for the refine load balancing strategy (Figure 5.15) and no load balancing strategy (Figure 5.13) we can see that load balancing becomes useful as the number of processors increase. Thus, to make things scalable in massively parallel systems it is important that the strategies for load balancing do not try to change the load drastically but should do it incrementally like in the refine strategy.

Chapter 6

Conclusions

We have presented here the design of a Adaptive Mesh Refinement library which provides the ability to write applications using 1D, 2D or 3D AMR. It implements the parallel tree data structure and abstracts the communication between processors from the user. Since this library is implemented in Charm++, the user can also take advantage of the load balancing framework in Charm++ to do dynamic load balancing. The main advantage of using this library is that the user has to implement code very close to sequential code, hence they do not have to deal with issues involved in writing parallel code. Another advantage of using this library is that it does not impose any restrictions on the data structures used by the user. Using a library is advantageous because using tested components makes the task of writing a big application easier and quicker.

6.1 Future Work

Quirk, in [27] said,

> *Often the effectiveness of a mesh refinement algorithm stems not from the sophistication of the components, which for the most part can be fairly mundane but from the way in which they can be combined as to overcome their individual weaknesses.*

I think with this in mind, the direction for future research in this area should be to build components that work together to maximize productivity. One of the important areas of research is to effectively visualize the data produced by the AMR applications. Another important area is to ensure the scalabilty of the components, which has become more critical as massively parallel systems are becoming more and more common. In this library, the control rests with the library which calls the user methods. Another approach for writing this library is using threads as used in the FEM Framework[3] developed at Parallel Programming Laboratary in University of Illinois at Urbana-Champaign .

References

[1] M J Berger and P Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys. 82*, pages 64–84, 1989.

[2] M J Berger and J Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys. 53*, pages 484–512, 1984.

[3] Milind Bhandarkar and L. V. Kalé. A Parallel Framework for Explicit FEM. In M. Valero, V. K. Prasanna, and S. Vajpeyam, editors, *Proceedings of the International Conference on High Performance Computing (HiPC 2000), Lecture Notes in Computer Science*, volume 1970. Springer Verlag, December 2000.

[4] Robert Brunner. *Versatile Automatic LoadBalancing with Migratable Objects.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 2000.

[5] Robert K. Brunner and Laxmikant V. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, February 1999.

[6] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. Technical Report 99-03, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999.

[7] David E. Culler and Jaswinder Pal Singh with Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1999.

[8] Message Passing Interface Forum. Document for a standard message passing interface. Technical Report CS-93-214, University of Tennessee, Nov 1993.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Weseley Pub. Co., 1995.

[10] R. Hornung and S. Kohn. The use of object-oriented design patterns in the samrai structured amr framework. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, pages 21–23, SIAM, Philadelphia, PA, October 1998.

[11] L. V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[12] L. V. Kale and Attila Gursoy. Modularity, reuse and efficiency with message-driven libraries. In *Proc. 27th Conference on Parallel Processing for Scientific Computing*, pages 738–743, February 1995.

[13] L. V. Kalé and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[14] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[15] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[16] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[17] L.V. Kalé and V. Saletore. Parallel state-space search for a first solution with consistent linear speedups. *Internaltional Journal of Parallel Programming*, 19(4):251–293, 1990.

[18] O. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of International Symposium on Computing in Object-oriented Parallel Environments*, Stanford, CA, Jun 2001.

[19] Xiangyang Li. Dynamic load balancing for parallel amr. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1999.

[20] Wissink A. M. and R. D. Hornung. Samrai: A framework for developing parallel amr applications. In *5th Symposium on Overset Grids and Solution Technology*, pages 18–20, Davis, CA, September 2000.

[21] Peter MacNeice, Kevin M. Olson, Clark Mobarry, Rosalinda deFainchtein, and Charles Packer. Paramesh : A parallel adaptive mesh refinement community toolkit. In *Computer Physics Communications*, volume 126, pages 330–354, 2000.

[22] W.F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. volume 15, pages 326–347, 1989.

[23] M.Parashar and J.C.Browne. Distributed dynamic data-structures for parallel adaptive mesh-refinement. In *Proceedings of the International Conference for High Performance Computing*, December 1995.

[24] Parallel Programming Laboratory, University of Illinois, Urbana-Champaign. *The Charm++ Programming Language Manual, Version 5.4*, April 1999.

[25] Parallel Programming Laboratory, University of Illinois, Urbana-Champaign. *AMR Programming Manual*, Nov 2001.

[26] T. Plewa and E. Mueller. Amra: A multidimensional adaptive mesh refinement hydrocode for astrophysics. In *Computer Physics Communications*, 2000.

[27] James J. Quirk. A parallel adaptive grid algorithm for shock hydrodynamics. volume 20, 1996.

[28] Stensland. Adaptive mesh refinement on structured grids. 1998.

[29] Trompert and Verwer. A static regridding method for two dimensional parabolic partial differential equations. 1991.

[30] Krishnan Vardarajan. Communication library for parallel architectures. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1999.

[31] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of Super Computing*, pages 12–21, November 1993.

Simple 2D AMR Apllication

A simple jacobi application using 2D AMR is presented in this section.

.1 Writing the .ci File

```
mainmodule jacobi2DAMR {
 extern module amr;
 extern module HeapCentLB;
 initcall void AmrUserDataJacobiInit(void);
 mainchare main
 {
   entry main();
 };
};
```

.2 Writing the .h File

```
#include "amr.h"
#include "HeapCentLB.h"
#define LB_FREQUENCY 15

class Jacobi2DAMR:public AmrUserData {
 private:
  int cellSize;
  double **dataGrid;
  double **newDataGrid;
  /*Utility Functions*/
  void copyGrid(void);
  void copyColumn(double *buf , int colNum) {
    for(int i=1; i<=cellSize;i++)
      buf[i-1] = dataGrid[i][colNum];
  }

  void copyRow(double *buf , int rowNum) {
    for(int i=1; i<=cellSize;i++)
```

```
      buf[i-1] = dataGrid[rowNum][i];
  }

  void copyToColumn(double *buf , int colNum) {
    for(int i=1; i<=cellSize;i++)
      dataGrid[i][colNum]=  buf[i-1];
  }

  void copyToRow(double *buf , int rowNum) {
    for(int i=1; i<=cellSize;i++)
      dataGrid[rowNum][i]= buf[i-1];
  }
  double sumofGrid(void) {
    double sum = 0.0;
    for(int i=1;i<cellSize+1;i++)
      for(int j=1;j<cellSize+1;j++)
sum += dataGrid[i][j];
    return sum;
  }

 public:
  /*Default Constructor: Called in the initial setup of the tree*/
  Jacobi2DAMR() {
    cellSize = 32;
    dataGrid = new double* [cellSize+2];
    newDataGrid = new double* [cellSize+2];
    for(int i=0;i< cellSize+2;i++) {
      dataGrid[i] = new double [cellSize+2];
      newDataGrid[i] = new double [cellSize +2];
      for(int k = 0;  k < cellSize+2; k++) {
newDataGrid[i][k]=10.0;
dataGrid[i][k] = (i+k) *1.0;
      }
    }
  }

  /*This constructor is called after refinement with data from te parent*/
  Jacobi2DAMR(void *data,int dataSize)
  {
    double *indata = (double*) data;
    cellSize = (int) sqrt((double) (dataSize/sizeof(double)));
    //    cellSize = cellSize/sizeof(double);
    dataGrid = new double* [cellSize+2];
    newDataGrid = new double* [cellSize+2];
    for(int i=0;i< cellSize+2;i++) {
      dataGrid[i] = new double [cellSize+2];
      newDataGrid[i] = new double [cellSize +2];
      for(int k = 0;  k < cellSize+2; k++) {
newDataGrid[i][k]= 10.0;
```

```
if(i== 0 || i == cellSize+1 || k==0 || k== cellSize + 1)
  dataGrid[i][k] = (i+k) *1.0;
else
  dataGrid[i][k] = indata[(i-1) * cellSize + (k-1)];
      }
    }

  }
  Jacobi2DAMR(CkMigrateMessage *m): AmrUserData(m){}

  PUPable_decl(Jacobi2DAMR);
  /*Mandatory Library Interface functions*/
  virtual void doComputation(void);
  virtual void **fragmentNborData(void* data, int* sizePtr);
  virtual void **getNborMsgArray(int *sizeptr);
  virtual void store(void* data, int dataSize, int neighborSide);
  virtual void combineAndStore(void **dataArray, int dataSize,int neighborSide);
  virtual bool refineCriterion(void);
  virtual void **fragmentForRefine(int *sizePtr);

  /*If load balancing is required*/
  virtual void pup(PUP::er &p);
  /*Destructor*/
  ~Jacobi2DAMR() {
    for (int i=0; i< cellSize+2;i++)
      delete [] newDataGrid[i];
    delete [] newDataGrid;
    for (int i=0; i< cellSize+2;i++)
      delete [] dataGrid[i];
    delete[] dataGrid;
  }
};

/*Main Chare*/
class main : public Chare {
 public:
  /*Constructor: Library is created from here*/
  main(CkArgMsg* args) {

    StartUpMsg *msg;
    msg = new StartUpMsg;
    msg->synchInterval = 200;
    msg->depth = 2;
    msg->dimension = 2;
    msg-> totalIterations = 500;
    CreateHeapCentLB();
    CProxy_AmrCoordinator::ckNew(msg,0);

  }
```

```
};
```

.3 Writing The .C File

```
#include "jacobi2DAMR.h"
#include "jacobi2DAMR.decl.h"
/*
********************************************
User Code for 2D
********************************************
*/
void AmrUserDataJacobiInit(void)
{
  PUPable_reg(AmrUserData);
  PUPable_reg(Jacobi2DAMR);
}

AmrUserData* AmrUserData :: createData()
{
  Jacobi2DAMR *instance = new Jacobi2DAMR;
  return (AmrUserData *)instance;
}

AmrUserData* AmrUserData :: createData(void *data, int dataSize)
{
   Jacobi2DAMR *instance = new Jacobi2DAMR(data , dataSize);
   return (AmrUserData *) instance;
}

void AmrUserData :: deleteNborData(void* data)
{
  delete [](double *) data;
}

void AmrUserData :: deleteChildData(void* data)
{
  delete [](double *) data;
}

void Jacobi2DAMR :: doComputation(void)
{
  for(int i=1; i <= cellSize ;i++)
    for(int j=1; j<=cellSize;j++)
      newDataGrid[i][j] = 0.2 * (dataGrid[i][j-1] + dataGrid[i][j+1]
           +dataGrid[i][j] +dataGrid[i-1][j] +
  dataGrid[i+1][j]);
  copyGrid();
```

```
}

void Jacobi2DAMR :: copyGrid(void)
{
  for(int i=0; i< cellSize+2; i++)
    for(int j=0; j<cellSize+2; j++){
      dataGrid[i][j] = newDataGrid[i][j];
      newDataGrid[i][j] = 10.0;
    }
}

void ** Jacobi2DAMR :: getNborMsgArray(int* sizePtr)
{
  //gives the size of each individual column in bytes
  *sizePtr = cellSize* sizeof(double);
  //since we are using 2D mesh so have an array of size 4
  double ** dataArray = new double* [4];
  for(int i =0;i<4;i++) {
    dataArray[i] = new double[cellSize];
  }
  //To my Right neighbor
  copyColumn(dataArray[0], cellSize);
  //To my left neighbor
  copyColumn(dataArray[1], 1);
  //To my Down neighbor
  copyRow(dataArray[2], cellSize);
  //To my Up neighbor
  copyRow(dataArray[3], 1);
  return (void **) dataArray;
}

void **Jacobi2DAMR :: fragmentNborData(void *data, int* sizePtr)
{
  int elements = (*sizePtr)/sizeof(double);
  int newElements = elements/2;
  double **fragmentedArray = new double* [2];
  double *indata = (double *)data;
  if(elements %2 == 0){
    *sizePtr = newElements * sizeof(double);
    for(int i=0; i<2; i++) {
      fragmentedArray[i] = new double[newElements];
      for(int j=0; j<newElements;j++)
fragmentedArray[i][j] = indata[i*newElements + j];
    }
  }
  else {
    *sizePtr =( ++newElements)*sizeof(double);
    for(int i=0; i<2; i++) {
```

```
      fragmentedArray[i] = new double[newElements];
      for(int j=0; j<newElements-1;j++)
fragmentedArray[i][j] = indata[i*newElements + j];
    }
    fragmentedArray[1][newElements-1] = indata[elements -1];
    fragmentedArray[0][newElements-1] = (fragmentedArray[0][newElements -2]
+ fragmentedArray[1][0])/2;
  }
  return (void **)fragmentedArray;

}

void Jacobi2DAMR :: store(void* data , int dataSize , int neighborSide)
{
  if(dataSize/sizeof(double) == cellSize) {
    switch(neighborSide) {
    case 0:
      copyToColumn((double*) data, 0);
      break;
    case 1:
      copyToColumn((double *) data, cellSize+1);
      break;
    case 2:
      copyToRow((double *) data, 0);
      break;
    case 3:
      copyToRow((double *) data, cellSize+1);
      break;
    }
  }
  else
    CkError("Error: Jacobi::store...wrong sized message size %d cellsize %d\n",
    dataSize/sizeof(double), cellSize);
}

void Jacobi2DAMR :: combineAndStore(void **dataArray,
int dataSize,
int neighborSide) {
  int size = dataSize /sizeof(double);
  double * buf = new double[2*size];
  double *tmpbuf = buf + size;
  memcpy((void *)buf, dataArray[0], dataSize);
  memcpy((void *)tmpbuf, dataArray[1], dataSize);
  store((void *)buf,(2*dataSize),neighborSide);
  delete []buf;
}

bool Jacobi2DAMR :: refineCriterion(void)
{
```

```
  double average = sumofGrid()/(cellSize*cellSize);
  if(average < 15.0 && cellSize >= 4)
    return true;
  else
    return false;
}

void** Jacobi2DAMR :: fragmentForRefine(int *sizePtr)
{
  int newSize = cellSize/2;
  *sizePtr = newSize*newSize*sizeof(double);

  double ** dataArray = new double* [4];
  for(int i=0;i<4;i++) {
    dataArray[i] = new double[newSize*newSize];
    for(int j=1;j<=newSize;j++){
      for(int k=1;k<=newSize;k++)
dataArray[i][(j-1)*newSize+(k-1)] =
dataGrid[((i/2)%2)*newSize+j][(i%2)*newSize+k];
    }
  }
  return (void **)dataArray;

}

void Jacobi2DAMR::pup(PUP::er &p)
{
  AmrUserData::pup(p);
  p(cellSize);

  //have to pup dataGrid and newDataGrid here
  if(p.isUnpacking()) {
    dataGrid = new double*[cellSize+2];
    newDataGrid = new double*[cellSize+2];
    for(int i=0;i<cellSize+2;i++) {
      dataGrid[i] = new double[cellSize+2];
      newDataGrid[i] = new double[cellSize+2];
    }
  }
  for(int i=0; i<cellSize+2;i++) {
    p(dataGrid[i],cellSize+2);
    p(newDataGrid[i],cellSize+2);
  }
}

PUPable_def(AmrUserData);
PUPable_def(Jacobi2DAMR);

#include "jacobi2DAMR.def.h"
```