PARALLEL INCREMENTAL ADAPTIVITY FOR UNSTRUCTURED MESHES IN TWO DIMENSIONS

BY

NILESH CHOUDHURY

B.Tech, Indian Institute of Technology, Kharagpur, 2003

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, 2006

Urbana, Illinois

Abstract

Many applications use unstructured meshes for solving a variety of problems in various fields of science and engineering. Most of these applications solve problems over irregular domains and tend to use unstructured meshing software. Applications could be brain models, climate models or engineering models to study material deformation and weapons strength. Apart from solid meshes, there exists a huge set of applications that solve computational fluid dynamics problems over meshes. Millions of lines of mesh framework source code exists to satisfy these applications. Most of these frameworks do not simultaneously support parallelism and geometric adaptivity - 'coarsening and refinement'.

Parallel adaptivity is a challenging problem of significance to application scientists. They would like to be able to adaptively give more importance to one part of the problem being studied, while keeping the computational requirements from blowing up. This requires the ability to refine parts of a mesh while at the same time coarsen other parts. The contribution of this thesis is a novel way to perform parallel adaptivity on large meshes. This thesis introduces an incremental method to perform parallel adaptivity. We define a set of primitive operations on a mesh and use these to design a set of atomic operations to perform adaptivity in parallel. Incremental operations give us a very low level tool for modifying the mesh. This gives a lot of power and flexibility to modify the mesh in any way one wants. The entire adaptive component is implemented as part of ParFUM, a parallel mesh framework. Moreover, scientific and engineering applications are successfully using or trying to use the parallel adaptivity presented in this thesis. These applications include engineering mechanics applications and Spacetime Discontinuous Galerkin applications.

To my parents who never cease to inspire me.

Acknowledgments

I would like to thank my advisor Prof. Laxmikant Kale for his invaluable advice, motivation and constant support without which this work would not have been possible. His ability to elaborate and provide infinitely valuable insight into absolutely any topic within a matter of mere seconds is absolutely amazing.

I would also like to thank my colleagues at the Parallel Programming Lab. Sayantan Chakravorty bore the brunt of most of my questions, be it Charm++, or ParFUM related or absolutely unrelated. I would also like to thank Terry Wilmarth for her constant help with most of the mesh adaptivity questions. Isaac Dooley also helped implementing adjacencies for entities in ParFUM. His help is much appreciated and invaluable to this thesis. I would like to thank Sandhya for using adaptivity for the physics applications. I am also thankful to all other people at the Parallel Programming Lab, who have made my stay here enjoyable, Yogesh, Amit, Filippo, CheeWai, David, Esteban, Abhinav, Eric, Gengbin, Yan, Tarun, Aaron.

Most of all I would like to give a heartfelt thank you to my parents, who have shown immense faith in my decisions. I am extremely grateful to my grandmother, my great grandmother, my younger brother and my sister. I draw immense strength and motivation from the wonderful family that I am lucky to be a part of.

Table of Contents

List of Tables . . . viii

List of Figures . . . ix

Chapter 1 Introduction . . . 1
1.1 Motivation . . . 1
1.2 Challenges . . . 2
1.3 Related Work . . . 3
1.4 Thesis Organization . . . 5

Chapter 2 Charm++, AMPI and Processor Virtualization . . . 6
2.1 Charm++ . . . 6
2.2 AMPI . . . 7
2.3 Multiphase Shared Arrays . . . 8

Chapter 3 Unstructured Meshing in Charm++ . . . 9
3.1 ParFUM: Parallel Framework for Unstructured Meshes . . . 9
3.2 ParFUM Communication . . . 11
3.3 ParFUM Ghosts . . . 12
3.4 ParFUM Interface to Applications . . . 13

Chapter 4 Parallel Adaptivity . . . 14
4.1 Primitive Parallel Mesh Modification Operations . . . 14
4.1.1 Add Node . . . 17
4.1.2 Remove Node . . . 18
4.1.3 Add Element . . . 19
4.1.4 Remove Element . . . 20
4.1.5 Purge Element . . . 20
4.2 Atomic Parallel Mesh Modification Operations . . . 21
4.2.1 Acquire Element . . . 21
4.2.2 Edge Flip . . . 22
4.2.3 Edge Bisect . . . 22
4.2.4 Edge Contract . . . 23
4.2.5 Longest Edge Bisect . . . 23
4.2.6 Vertex Remove . . . 24

4.2.7 Vertex Split . . . 25
4.3 Mesh Adaptive Algorithms . . . 25
4.4 Managing Ghost layers . . . 27

Chapter 5 Parallel Adaptivity Implementation in ParFUM . . . 29
5.1 Maintaining Adjacency Information . . . 30
5.2 Prioritized Locking . . . 31
5.3 Mesh Boundary . . . 32
5.4 Mesh Corners . . . 33
5.5 Flexible Boundaries Between Chunks . . . 34
5.6 Discontinuous Chunks . . . 35
5.7 Empty Chunks . . . 36
5.8 Node from Local to Ghost . . . 36
5.9 Solution Transfer . . . 37
5.10 Preserving Mesh Geometry correctness . . . 38

Chapter 6 Parallel Adaptivity Results and Performance Analysis . . . 40
6.1 Parallel Adaptivity Results . . . 40
6.2 Parallel Adaptivity Performance Analysis . . . 42
6.2.1 Benefits from Medium-Sized Chunks . . . 42
6.2.2 Benefits from Virtualization . . . 44
6.3 Parallel Performance Scaling . . . 45
6.4 Profiling Parallel Adaptivity: Where does the Time go? . . . 51
6.5 CPU and Memory Performance Optimizations . . . 52

Chapter 7 Applications using Adaptivity . . . 54
7.1 1D Wave Propagation Problem . . . 54
7.2 Generating a Billion Element Mesh by Continuous Refinement . . . 57
7.3 Spacetime Discontinuous Galerkin . . . 60

Chapter 8 Conclusion and Future Work . . . 61

References . . . 63

List of Tables

6.1 Profiled Information for Low Level Operations in Mesh Adaptivity 51
6.2 Profiled Information for Mesh Adaptivity Abstract Functions 51

List of Figures

2.1 The Charm++ runtime system maps virtual processors to physical processors 7

3.1 Pseudo-code for a typical ParFUM application 13

4.1 Chunk 2 acquires e1 from Chunk 1 21
4.2 Edge flip (n1,n2) 22
4.3 Edge bisect (n1,n2) 22
4.4 Edge contract (n1,n2) 23
4.5 Longest Edge Bisect (n1,n2) 24
4.6 Vertex Remove (n5) 24
4.7 Vertex Split (n1) 25

5.1 Mesh Boundary Idiosyncracies 33
5.2 Need for fluid chunk boundaries 35
5.3 Transition from local to ghost node 37
5.4 A Flip Destroying Mesh Geometry 38

6.1 A set of adaptive operations on a 750-node square mesh on 6 processors . . . 41
6.2 A set of adaptive operations on a 88-node L-shaped mesh on 3 processors . . 42
6.3 10 sets of adaptive operations on a 13k element mesh across 6 processors . . 43
6.4 A single refinement operation on a 240k element mesh across 12 processors . 44
6.5 A single coarsening operation on a 240k element mesh across 12 processors . 44
6.6 A single coarsening operation on a 240k element mesh across 12 chunks but 2 processors 45
6.7 Demonstrate Virtualization benefits on Problem 1 for 8 and 16 processors . . 46
6.8 Speedup for Problem 1 at different virtualizations 47
6.9 Speedup for Problem 2 at different virtualizations 48
6.10 Speedup for Problem 1 at best performance and fixed number of VPs 49
6.11 Speedup for Problem 2 at best performance and fixed number of VPs 50

7.1 Shock-wave on a bar 55
7.2 Mesh and velocity for Shock-wave on a bar on 8 processors 55
7.3 Mesh for Shock-wave on a bar on 8 processors 56
7.4 Velocity for Shock-wave on a bar on 8 processors 56
7.5 Scaling problem size/processor with time at virtualization of 8 59

Chapter 1

Introduction

1.1 Motivation

Electrical engineers use Maxwell's equations to simulate electric and magnetic fields in motors, radio antennae, silicon chips and even space satellites. Civil and Mechanical engineers study stress-strain relationships to simulate impact and failure response of structures, be it cars, aero-planes, rockets, space shuttles, buildings or bridges. Climate modellers use wave equations to represent atmospheric and ocean currents to study different climate phenomenon. Theoretical cosmologists use relativity to simulate space-time gravity waves emanating from stars. Weapons developers use atomic physics over the nuclear stockpile to study its health. All of these applications depend on computational science.

Most important to computer scientists, each of these applications rely on a set of services that provide the mesh structure on top of which the application level physics is built. Each of these applications present a different challenge for computation and communication load imbalance. It would be extremely relevant to application scientists if all these mesh related services are abstracted from them and provided in parallel reliably through a framework which they can interface their physics dependent algorithms to. In addition, parallel computing is hard for computer scientists, so to expect application scientists to understand and write efficient parallel code is not fair.

Apart from maintaining basic mesh geometric functionality, application scientists would

like to have adaptivity built into the geometric meshes they are studying. They would like to study different regions of their problem domain at different levels of detail at different times. A higher level of detail would mean that they need more elements to apply physics on, than initially existed in some region. This necessitates the framework to provide refinement services to the application. However, this leaves the computer scientist with a grave problem. Computation power is neither inexpensive nor available in infinite supply. When the application scientist directs his attention to some other part of the problem, the computer scientist has to make sure that the now 'not so important' regions are turned back to a normal level of detail, so as not to waste computation power. This involves the most difficult component of mesh adaptivity - 'the ability to coarsen a region of the mesh efficiently in parallel'. Add to this the fact that meshes will have ghost layers on their boundaries with different chunks and the boundaries can be arbitrary - meaning they will not necessarily have any structure. The application, might at the same time need to refine some region and coarsen some other region of the mesh. All this makes the problem interesting to computer scientists and the benefits invaluable to application scientists.

1.2 Challenges

Different applications would show absolutely different and idiosyncratic characteristics as far as the computer scientists are concerned. Climate modelling applications would like to hone into a region of turbulence and study it more carefully; fracture studies would like to study in more detail the tip of a crack or the point of impact, because that would naturally give them more insight. For each application this translates to different types and regions of adaptivity; different quality metrics would be essential for different applications as the physics they apply is different and for that physics, elements of different shapes might be more amenable.

- The biggest challenge here is undoubtedly to implement refinement and coarsening in

parallel. This means correctness should be ensured and the mesh should be consistent as defined in section 4.1 after the modifications.

- Since this is computational science, we have to ensure an efficient implementation, so that users can solve larger problems with the minimum available resources – leading to maximum productivity of the resources.

- Computation and Communication load can widely vary between different regions of the mesh making it imperative to provide efficient load balancing if resources are to be used anywhere near their optimum potential. This can be done at two levels. The first comes naturally by using processor virtualization. A processor that has an overloaded chunk (virtual processor), can shed off some other chunks to under-loaded processors. At another level, an overloaded chunk can shed off some elements that belong to it, to neighboring chunks. These will be explained in more detail later in Section 5.

- It is also important and challenging to preserve the shape of a mesh while performing different adaptive operations on the mesh. Refinement usually does not try to change the shape of the mesh, but coarsening if not done intelligently can break the shape of a mesh at physical boundaries.

- Another challenge is to maintain the ghost layers of different chunks while performing adaptivity, as the elements on the boundary change or the boundary itself between two chunks starts moving.

1.3 Related Work

The need for a framework that would provide parallel unstructured framework services that would harness the compute power of machines and provide an adaptive set of mesh modification functions and abstract this as a set of services to the application scientist has been felt by the community for a long time. But most frameworks accessible are either aimed

at structured meshes or do not work in parallel. Simpler frameworks exist in abundance which use simple data structures to even provide unstructured mesh adaptivity but they are mostly non-parallel. A lot of complications arise when we want these adaptive operations on unstructured meshes in parallel. So, the complications arise from a combination of two important but absolutely essential requirements – support for unstructured mesh and parallel operations. Here we will not talk about serial mesh adaptive frameworks. Instead we will present some of the attempts to provide adaptivity in unstructured meshes in parallel.

The first framework that inevitably needs a mention is SIERRA being developed at the Sandia National Laboratories. While, it is not yet publicly available, it is expected to have a huge feature set. It promises a generic framework that would provide adaptivity. But according to the latest accessible publication [15], the adaptivity it supports is hierarchical and not completely geometric.

Simmetrix is a commercial software package that targets the entire mesh simulation workflow. However, it mostly provides non-parallel features. Some features are being modified to support parallelism. A commercial version exists with some parallel adaptivity operations.

RPI [5] presents another framework that provides parallel mesh adaptivity, ParFUM [13] has mesh modification primitives very similar to these.

PREMA [2] is a runtime specially created for parallel mesh generation applications. It has introduced adaptive runtime features recently that Charm++ has supported for a long time.

LibMesh is a C++ library that provides a bunch of mesh modification functions. This library supports some parallel operations on SMP machines or on clusters via MPI. It supports some refinement operations, but is limited to hexahedral elements.

CGAL is geometric library which provides a huge number of geometric operations that can be used in meshing adaptivity, but this work is serial.

1.4 Thesis Organization

In Chapter 2, we give an introduction to the existing virtualization framework which we leverage for our work. It also gives an insight into the different technologies including AMPI and TCharm on which ParFUM is built. The next chapter 3 gives an introduction to the unstructured meshing framework. It presnts the language interface that ParFUM provides to an application as well as the index list communication library. The following chapter 4 explains the parallel mesh modification operations that we introduce and the concept of primitive operations and atomic operations on a mesh. We also give a brief description of some of the adaptive algorithms which use these set of operations and provide an interface to bring adaptivity to the application scientist. Chapter 5 presents and explains some of the challenges faced while implementing these operations to work in parallel. We also explain locking, boundary preservation and solution transfer. Chapter 6 presents a study of the performance of the adaptive part of the framework. It also presents an insight into some optimizations and the benefits obtained from the optimizations. Chapter 7 presents some applications which have been built on ParFUM and have already used the adaptive component successfully. It also talks about some applications that are actively trying to use this capability. Finally, Chapter 8 presents a summary of conclusions for this thesis. It also presents some thoughts into future possible research that would enhance the capabilities of this work.

Chapter 2

Charm++, AMPI and Processor Virtualization

ParFUM [13] is based on Charm++, hence it inherits all its capabilities - dynamic load balancing; check-point and recovery; communication optimizations; etc. It also gains the advantage of being naturally able to run on a vast set of architectures that Charm++ supports. In this chapter, we present an introduction to Charm++, AMPI, TCharm and MSA. These are the underlying technologies, and ParFUM leverages a lot of features from them.

2.1 Charm++

Processor virtualization [11] is the key idea behind Charm++[10]. The user breaks up his computation into a large number of objects without caring about the number of physical processors available. These objects are referred to as *virtual processors*. The user views the application in terms of these virtual processors and their interactions. The Charm++ runtime system allows the virtual processors to interact among each other through asynchronous method invocation. Figure 2.1 shows the users view of an application and the system view of it.

The Charm++ runtime system also maps the virtual processors to physical processors. The user code is independent of the location of a virtual processor. This allows the runtime system to change the mapping of virtual processors to physical processors in the middle of

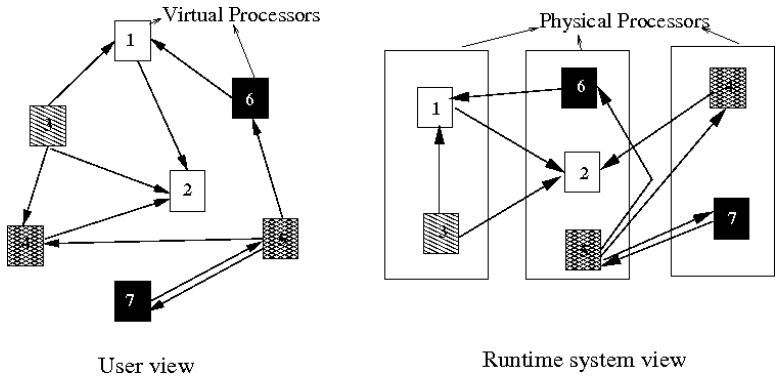

Figure 2.1: The Charm++ runtime system maps virtual processors to physical processors

an execution. It *migrates* one virtual processor on one physical processor to another physical processor at runtime. The Charm++ runtime system allows for message delivery and collective operations such as reductions and broadcasts in the presence of migrations. So the runtime system can migrate objects away during an execution to adapt to the changing load characteristics of a problem. If at some point in the execution some objects on one processor start doing more work, the runtime system can distribute them uniformly among all the other physical processors. Thus processor virtualization enables us to perform measurement-based run-time load-balancing. Distributed and centralized load balancers can be developed to remap virtual processors to physical ones. Runtime load balancing has been used to scale molecular dynamics to thousands of processors [12].

Another major benefit of processor virtualization is the automatic adaptive overlap between computation and communication. If one virtual processor on a processor is waiting for a message, another one can work on the same physical processor.

2.2 AMPI

Adaptive MPI (AMPI) [8, 4] is an implementation of the Message Passing Interface (MPI)[14, 7] on top of Charm++. Each MPI process is a user-level thread bound to a Charm++ virtual processor. AMPI derives its implementation from a threaded version of Charm++ where

each Charm++ object has just one thread of execution, and these threads are migratable. Threaded Charm (TCharm), provides a common runtime support for all threads and allows threads from different frameworks to communicate among them. The MPI communication primitives are implemented as communication between the Charm++ objects associated with each MPI process. Traditional MPI codes can be used with AMPI with slight modification. These codes can also take advantage of automatic load balancing and adaptive overlap of communication and computation.

2.3 Multiphase Shared Arrays

Another feature of Charm++ used in ParFUM is the Multiphase Shared Array (MSA)[6]. MSA is used extensively in the parallel partitioning component of ParFUM. MSA is a distributed shared memory model in which data is accessed in phases. In each phase all the data elements in a particular array are accessed in the same mode by all participant threads. The mode of an array can be changed between phases. The different modes supported by MSA are read-only, write and accumulate. In the *read-only* mode participants can read as many elements as they want. The *write* mode allows only one participant to write to any particular data element. In the *accumulate* mode, a commutative-associative operation is used to accumulate data contributed by different participants to a single element.

MSA data elements can be load balanced at run time on the basis of computation and communication load so that data elements move to the processors where most of their accesses originate. MSA can also be used to read in a huge amount of data on one processor but store it simply and efficiently on several processors.

Chapter 3

Unstructured Meshing in Charm++

This chapter describes the basic infrastructure of ParFUM. It describes the concepts in ParFUM. We explain the usage of the framework along with the adaptive component from the user's perspective. We also explain ParFUM communication between chunks and ghost support in ParFUM [13].

3.1 ParFUM: Parallel Framework for Unstructured Meshes

The precursor of ParFUM was the Finite Element Framework [3]. However, ParFUM goes beyond its precursor by supporting:

- parallel mesh adaptivity - geometric coarsening and refinement
- parallel mesh partitioning
- detailed boundary conditions
- user code to mix in arbitrary MPI parallel communication in between mesh computation. Both MPI and AMPI are supported.
- a simple interface that works with C++ as well as Fortran90.

The ParFUM framework is written on top of MPI. Its structure is a good fit for the MPI style of programming. Although it works on all MPI versions, using it with AMPI provides the advantages of processor virtualization. Load balancing is especially useful for applications with refinement where the amount of computation on a particular chunk can change significantly as the number of elements on it varies, which is a very common side-effect of mesh adaptivity. It is also helpful in applications where the computation load for some elements varies at different stages of the simulation.

In this section, we describe in brief some of the basic infrastructure that make up ParFUM [13].

- *Domain* The space in which the user is trying to solve a problem. E.g. When simulating stress wave propagation in a bar, the domain is the bar.

- *Node* An individual point in the problem domain. Nodes always have at least coordinate, valid and boundary data associated with them, and often include various solution data as well.

- *Element* A small piece of the problem domain, defined by the nodes surrounding it.

- *Node Adjacencies* For each node, we maintain at all times information about the set of nodes it is connected to. This is called the node-to-node adjacency. ParFUM also maintains a list of node-to-element adjacency.

- *Element Adjacencies* For each element, there is element-to-node connectivity and element-to-element connectivity. The former defines an element uniquely, while the latter is maintained to help perform the adaptivity operations faster.

- *Solution Data* Data associated with various entities in a mesh that represent some step in the problem solution process. In ParFUM, solution data is represented as an *attribute* of the corresponding *entity*, as described below.

- *Mesh* A group of nodes, elements, solution data, and boundary conditions that together represent a problem domain. It supports operations to adapt a region of the mesh or modify a region to a required quality metric.

- *Chunk* One partition of a mesh. Normally each MPI process has exactly one chunk of the mesh, although in AMPI, multiple MPI processes are present on each processor.

- *Entity* A generic term for anything in a mesh that can hold data: a node , an element, a sparse element or ghost nodes, ghost elements.

- *Attribute* An array of solution data associated with an entity.

ParFUM presents a simple set of operations to load a mesh, associate data with each entity in the mesh, partition the mesh and then start solving some physics on the mesh.

3.2 ParFUM Communication

This satisfies an application's parallel communication needs. For a mesh partitioned among several chunks, there are two types of communication that needs to be supported - ghost (nodes and elements) and shared nodes. These are implemented through an Index List Communication Library (IDXL). The implementation of IDXL lists maintains a consistent list between every pair of chunks that share an entity for each entity. So, if chunk 'A' shares nodes with chunk 'B', then both of them will have a shared IDXL list for each other, where each maintains the index number of the attribute it shares with the other chunk in a consistent order. Ghosts are treated slightly differently. Suppose, chunk 'C' sends a ghost to chunk 'D', then chunk 'C' will maintain a sendghost list for chunk 'D', while chunk 'D' in its turn will maintain a receiveghost list for chunk 'C'. The receive ghost IDXL list is associated with the ghost entity, while the send ghost IDXL list is associated with the real entity. In addition any entity can have only one IDXL list for every chunk it needs to communicate to.

Applications mostly perform some kind of calculation in which each node or element requires data from all of its neighboring elements / nodes. This implies that if a node is on the boundary or if an element has an edge on the boundary, then it requires some data from another chunk. Instead of requesting data for each individual element / node every single time, we have the concept of ghosts, which is a read-only copy of a real entity that exists on a neighboring chunk. ParFUM provides a single call to update the read-only ghost entity data with the actual entity. This allows applications to seamlessly access data across chunks with minimum communication overhead. ParFUM directly supports communication over shared nodes, ghost nodes and ghost elements equally well. The number of messages is always exactly equal to the number of adjacent mesh chunks, i.e. the number of IDXL lists and the amount of data exchanged is almost always approximately equal too.

If application employs adaptivity, the solution data on each entry associated with an attribute is modified as the entry in an entity is updated. This updation is finished before the adaptive call returns back to the user. We support default set of operations to distribute the data on an element across the new set of elements that replace it. The data on nodes is also interpolated or copied across nodes, when they are created during refinement. The user can override these functions to change different solution data differently as the mesh is modified.

3.3 ParFUM Ghosts

As shown in Section 3.2, ParFUM adds ghost elements and nodes to allow the elements along a chunk boundary to access data from neighboring elements on another chunk seamlessly. ParFUM allows applications to define ghosts. For one application two tetrahedra that share an edge might be considered neighbors, whereas in another case only tetrahedra that share faces might be considered neighbors. The user can also have multiple layers of ghosts for applications that need neighbors of neighbors.

3.4 ParFUM Interface to Applications

A ParFUM application looks very much like a typical MPI application. At start-up, the mesh is read in. Users have their choice of reading their existing serial mesh on processor 0 and partitioning on the fly, or partitioning the mesh beforehand and reading the mesh partitions in parallel in ParFUM format. Once the mesh is set up, the program executes some sort of solution loop over time or iterations (or other work) for the local chunk of the mesh, occasionally communicating with other chunks.

As the solution is being calculated on the mesh, the user can associate various quality metrics with each individual element across the entire mesh. As computation progresses, the user can change the requirements for the quality metrics and make a call to ParFUM to adapt the elements to adjust to the new set of element quality requirements. The user could be oblivious about which processor these sets of elements belong to, or even if they belong on the boundary of two chunks and the adaptive operations will be performed irrespectively.

```
1. read the mesh and configuration data (this part is mostly serial)
2. get local mesh chunk
3. set mesh quality criteria
4. time loop
5.   perform adaptivity to get the desired mesh
6.   perform physics calculations in this time-step
7.   communicate boundary conditions
8.   more physics computations
9. end time loop
```

Figure 3.1: Pseudo-code for a typical ParFUM application

Chapter 4

Parallel Adaptivity

The motivation to provide adaptivity in a meshing framework is immense. Overall adaptivity can provide better numerical accuracy and more insight in simulations with minimal increases in computation expense. Providing support for adaptivity of meshes in parallel is not as simple as it would be for a mesh on a single processor. Moreover, some applications need to perform incremental and unsynchronized modifications to their parallel meshes. One application that needs parallel mesh modification is the adaptive space-time meshing algorithm[1, 9], described in Section 7.3. To support different forms of mesh modification in ParFUM, we decided to create a simple but robust abstraction that would support a wide range of application needs. The modifications included in this ParFUM extension are not limited to subdivision of existing elements, as is the case in many meshing packages that support adaptivity. The mesh modification functions in a ParFUM application can be performed at any time, without any global synchronization.

4.1 Primitive Parallel Mesh Modification Operations

The importance of asynchronous mesh modification suggests that no communication needs to be done between chunks on which the region of the mesh being modified has no elements associated with. The baseline being, if an element is being modified, then only chunks that know about this element need an update. The chunks that know about this element are:

- the chunk that owns this element (there can be only one such chunk).
- all the chunks that know about this element, i.e. have this element as a ghost.

For an element e_i, let the function $K(e_i)$ determines the set of chunks that know about this element. $O(e_i)$ is the chunk that owns this element, while $G(e_i)$ is the set of chunks that have this element as a ghost. Thus, if we want to modify a region which contains a set of elements that belong to the set S. The only chunks that need to be involved in the operation is obtained from $\sum_{e_i \in S} K(e_i)$. Thus, ParFUM adaptivity needs to synchronize only the chunks that need to perform the adaptivity operation. Other chunks can go about performing their computations. This we believe is extremely important, since most of the situations where adaptivity is expected to be used will have a large mesh spread over hundreds of processors, but adaptivity at a time would be required only in a specific region, which might involve just one processor or only a few processors at most.

Another important characteristic of ParFUM is its ability to allow users to perform incremental operations on any element of the mesh. Users have the ability to specify exactly which element they want modified, and what exactly they want to be done to that element. This we believe is an extremely powerful tool in the hands of the user. However, we realize that the user might not be interested in this level of power. ParFUM also provides an abstract function where the user can specify some quality metric for elements and the framework will try to maintain the quality metric for these elements. Users can update the quality requirement at certain regions and call the mesh modification routine to force a region of the mesh to refine or coarsen or perform a set of refinements and coarsenings alternately as the framework best decides to, in order to achieve a desired final quality requirement. Apart from this, the framework also makes sure to preserve element quality, as adaptivity can easily generate elements that are sub-standard and can cause numerical errors (e.g elements with very small areas, or super thin elements, also called slivers).

Here, we describe the set of operations we defined at the lowest level that allows the framework to perform any adaptive operation. ParFUM design is based on a layered approach to make the design simple and implementable. It is based on the principle of "divide and conquer". The problem of supporting adaptivity in parallel is broken down into supporting the following set of primitive operations in parallel. As described in Section 3.1 a mesh is made up of entities - nodes and elements. To support adaptivity we need to support the concept of modifying these basic entities – i.e. add and remove nodes and elements in parallel.

Before we delve into the adaptive primitive operations, we need to describe some important terms referring to the state of a mesh being modified.

- *Consistent Mesh*: A mesh is said to be in a consistent state if all nodes and elements that belong to the mesh are in a consistent state.

- *Consistent Node*: A node is said to be consistent if it has a consistent set of connectivity. A node n_i is connected to a set of other nodes $N(n_i)$ and each of these connections should be an edge of a valid element. Moreover, $N(n_i) \notin \phi$. n_i also has an element connectivity $E(n_i)$, which is the set of all elements that have this node in their connectivity table, or $element-to-node$ adjacency table. Geometrically $E(n_i)$ is the set of elements that surround this element. Also, $N(n_i) = \cup_{e_j \in E(n_i)} N(e_j)$. Other validity constraints include, all elements in $E(n_i)$ completely surround the node n_i if none of the elements are on the actual boundary of the mesh.

- *Consistent Element*: An element is said to be in a consistent state, if it has a consistent connectivity. i.e. it is connected to consistent nodes. $N(e_i)$ is the nodal-connectivity of an element. $E(e_i)$ is the set of elements that share an edge with e_i. Each of these elements should in turn be valid. The definition of validity of an element is slightly fuzzy in the sense that it is considered to be invalid after it loses all its connectivity, though it still exists in the invalid state for some time before it is deleted. This is

important as we want to replace an element's solution data and only after we copy it out to another element that will be replacing this element, we finally delete the element. For the discussion here, once an element loses all its connectivity, it is considered to be invalid.

Note that the above set of definitions for consistent nodes and elements only refers to local elements and shared/local nodes. It does not refer to ghosts. Ghost nodes and elements have a similar validity criteria, but they do not necessarily have a complete connectivity table, because some of the connectivity of a ghost node or element could have elements and nodes that the local chunk does not know about. Hence, if we consider all nodes and elements, including local and ghost on one chunk as forming a mesh, on this chunk which has the outermost set of elements as forming a "physical boundary", then all the above definitions are valid for them too. An important criteria that the mesh should always conform to, is that at all times except in the middle of atomic operations, all processors should have the same view of whatever region of the mesh they know about as a local or ghost entity.

4.1.1 Add Node

There are three types of nodes on a chunk:

- *local node*: A local node which no other chunk knows as local on it.
- *shared node*: A local node that some other chunk(s) might know as local on them.
- *ghost node*: A non-local node that some other chunk(s) might know as local on them.

A chunk might want to add a local, shared or ghost node.

- If it wants to add a local node, then it must have originated the process. A local node could only be created by interpolating it among a bunch of existing nodes. In 2-D geometry, this usually means that a node could be added between two nodes that form

an edge, or between three nodes that form a face. Though technically more is possible, we will need to only add nodes between edges, for purposes of refinement.

- If a node is being added between two shared nodes, only then is it a shared node. In this case, we need to figure out all the chunks that share this edge. In 2D, it can be noted that only two chunks could share an edge, though this is not true in higher dimensions. Let $S(n_1)$ is the set of chunks that share $n1$. $S(n_2)$ is the set of chunks that share n_2. Now, $S(n_1) \cap S(n_2)$ is the set of chunks that the new node should be added on. Also, every pair of these set of chunks, need to add this new node in the shared node (SN) index list.

- A ghost node would be added to a chunk, if a new ghost element, g_i is being added to a chunk and this new element has a node in $N(g_i)$, which this chunk does not know of. In this case a new ghost node is created on this chunk and it is mapped to this node in its ghost node receive (GNR) index list. The originating chunk which is sending this new ghost node, should add this node in its ghost node send (GNS) index list.

The local and shared nodes maintain all the data attributes for a node. So, when a new node is created, it generates all the data by interpolating values from the nodes from which it is created. For ghost nodes, chunks do not maintain any solution attributes.

4.1.2 Remove Node

A node is allowed to be removed only when it has been stripped of all connectivity. When a mesh is being modified, it normally goes about by adding nodes, removing existing elements and adding new ones. In this process, if it wants to delete a node n, it has to first get rid of n in the $N(e_i)$ of all elements e_i. Then removing a local node is just a matter of making it invalid before verifying it has no connectivity. A shared node should only be deleted, if has no connectivity on all shared chunks. A ghost node deletion actually means a totally different thing. It just means that the ghost instance on this node is being removed. So, this

element on the ghost node entity is made invalid, after making sure it has no connectivity on this local chunk. A chunk might be removing a ghost node if it has lost some ghost element that was causing that node to be a ghost on this chunk. The chunk that owns this node might or might not still possess it..

4.1.3 Add Element

This is one of the most complicated operations. It should be noted that an element can only be

- *local*: An element can only be local on one chunk. This chunk owns this element. All nodes $n_j \in N(e_i)$ are local/shared.
- *ghost*: An element can be a ghost on a number of chunks. Only one chunk can send an element as a ghost. This chunk needs to put this information in the ghost element send (GES) list of all chunks it sends this ghost to. All the receiving chunks needs to add this information in the ghost element receive (GER) list. For a ghost element e_i, $n_j \exists N(e_i)$, where n_j is a ghost node.

When an element is being added, the first thing computed is how many of the nodes in its connectivity is local, shared or ghost.

- if all nodes are local and not shared, we just need to add a local element and this element will not be a ghost anywhere, so no ghosts should be built.
- if at least one node is a shared node, and at least one node is a local node, then this element needs to be added locally, while ghosts on other chunks need to be built.
- if all nodes are shared, we need to decide which chunk should own the element, some suggestion could be given by the higher level function that calls this function, otherwise, it can decide based on connectivity. Then it needs to build ghosts on the concerned chunks.

- if any node is a ghost, then this element needs to be added on some other chunk. We decide which chunk it should be added locally to, and add it locally on that chunk. That chunk then does all the work. i.e. one of the above conditions gets satisfied on that chunk and ghost building originates there.

Building ghosts is also an involved topic and will be discussed in section 4.4.

4.1.4 Remove Element

Removing an element is also a very complicated operation. The framework figures out if an element to be removed is a local or ghost element. The various cases involved here are very similar to the previous subsection. Depending on the type of nodes in the connectivity of the element, the chunk owning the element is asked to remove the element. The local chunk invalidates the element after disconnecting all connectivity on adjacent nodes and elements. Then it goes about building ghosts if needed on all concerned chunks. Deciding the set of chunks where ghosts need to be built is not very complicated, but building the ghosts is. It will be discussed in section 4.4.

4.1.5 Purge Element

An element is deleted in two stages. It exists until it is purged. There is a state when it exists, even thought it has no connectivity and is geometrically invalid. This is to support copying or deriving the solution data from this element to the new element that replaces this. This is because, we always maintain a consistent set of connectivity, so, we have to delete an element before we add an element that replaces this. So, connectivity is always first removed and then added. Temporarily, the mesh has a fuzzy geometry, but a remove is always followed by an add and a lock is kept during these operations, so that at the end of the atomic operation, the mesh is consistent.

Removing local ghost elements is comparatively easier and a local operation, it is removed

only after ensuring that all local connectivity to this element as far as this chunk is concerned has been deleted.

4.2 Atomic Parallel Mesh Modification Operations

This section describes a set of atomic operations, which are used to provide mesh modification operations. To understand these operations, we need to understand that these operations are atomic in nature. To ensure atomicity of we need to lock the concerned nodes over which this operation is defined, so that the mesh is in a consistent state before and after the operation. Locking also ensures that no other operation gets performed in this region, when one chunk is already working on these set of nodes. More information on locking is provided in the section 5.2.

4.2.1 Acquire Element

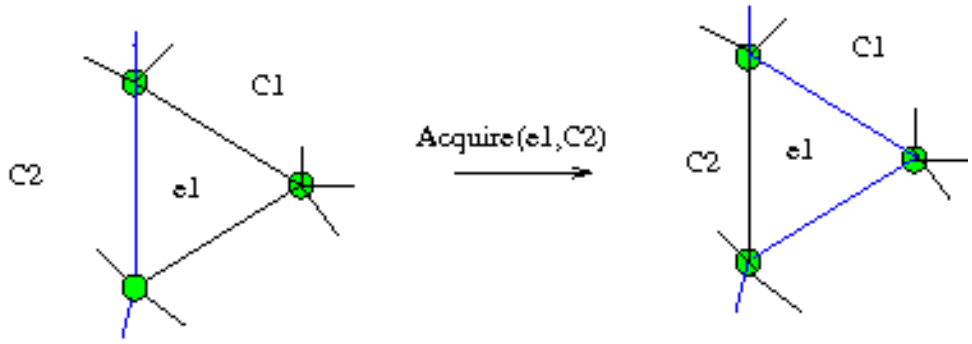

Figure 4.1: Chunk 2 acquires e1 from Chunk 1

This operation is defined on an element e_1 on a chunk boundary. One needs to specify the chunk C_2 which wants to acquire this element. This element is local on C_1. This element should have one edge which is shared between the chunks C_2 and C_1. The operation involves locking all nodes $n_i \in N(e_1)$. Then the element is removed from chunk C_1 and added to chunk C_2. Finally all the locks acquired are released. This operation is used to move the

chunk boundary, which is needed in several coarsening operations or in offloading elements from a chunk.

4.2.2 Edge Flip

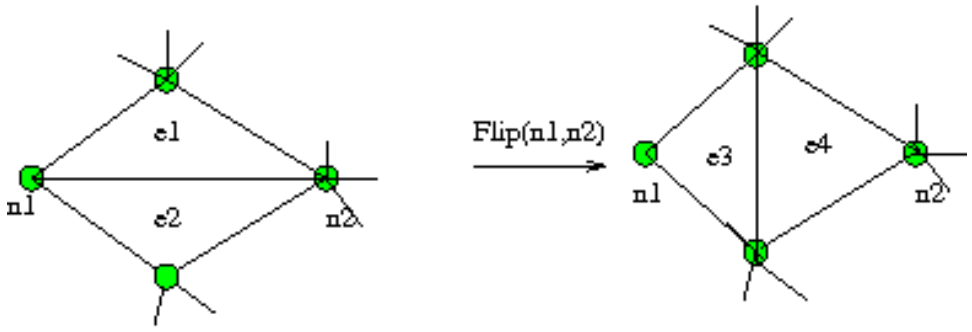

Figure 4.2: Edge flip (n1,n2)

This mesh modification operation is used to improve element quality. It is defined on an edge, i.e. the two nodes n_1 and n_2 which form an edge. We find the two elements e_1 and e_2, which share this edge. Then we lock nodes $N(e_1) \cup N(e_2)$. Next, we remove element e_1 and e_2 and add two new elements e_3 and e_4, with modified connectivity.

4.2.3 Edge Bisect

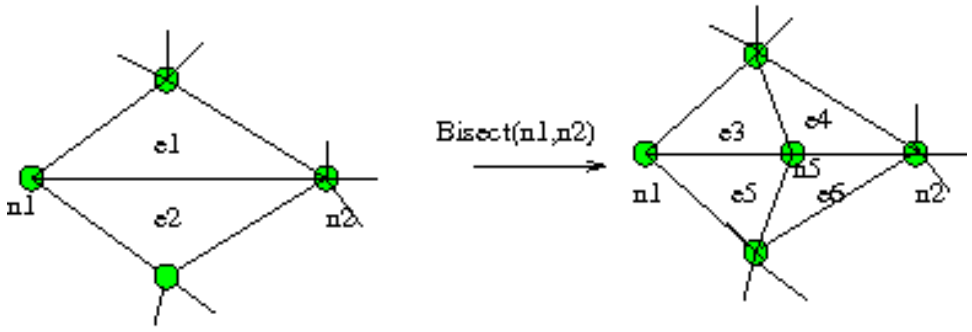

Figure 4.3: Edge bisect (n1,n2)

This mesh modification operation is performed to refine a region. It creates new elements whose area is half of the area of the previous elements they replace. The overall operation

is similar. Lock involved nodes. Add a new node $n5$. Remove elements $e1$ and $e2$, and add elements $e3$, $e4$, $e5$ and $e6$. Finally, we unlock all locked nodes.

4.2.4 Edge Contract

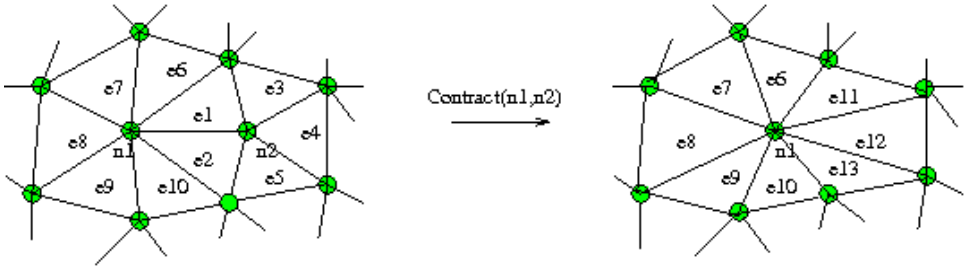

Figure 4.4: Edge contract (n1,n2)

This mesh modification operation is performed when we want to increase the area of elements in a region. In figure 4.4, the edge $n1$, $n2$ is to be contracted. This involves locking all the surrounding nodes. The elements $e1$ and $e2$ are removed. The coordinates of $n1$ are updated to the new location, interpolated between $n1$ and $n2$. Now, all elements connected to $n2$, are also removed. These are elements $e3$, $e4$ and $e5$. They are now replaced by the elements $e11$, $e12$ and $e13$. However, before this operation is performed, we make sure that the geometry of the element is still valid. It is important to ensure we do not generate bad elements (slivers), or worse, completely bad geometry in the region. More detail about this will be given in a later section. Finally we unlock all locked nodes.

4.2.5 Longest Edge Bisect

This is also a refinement operation, however, the refinement propagates, till the edge being bisected between two elements is the longest edge on both the elements. This operation is described on an element. The longest edge for this element is chosen to be bisected. In figure 4.5, we do the operation longestEdgeBisect(e2). We find the longest edge of $e2$. This is the edge $(n1, n2)$. The two elements on two sides of it are $e1$ and $e2$. So, we bisect this

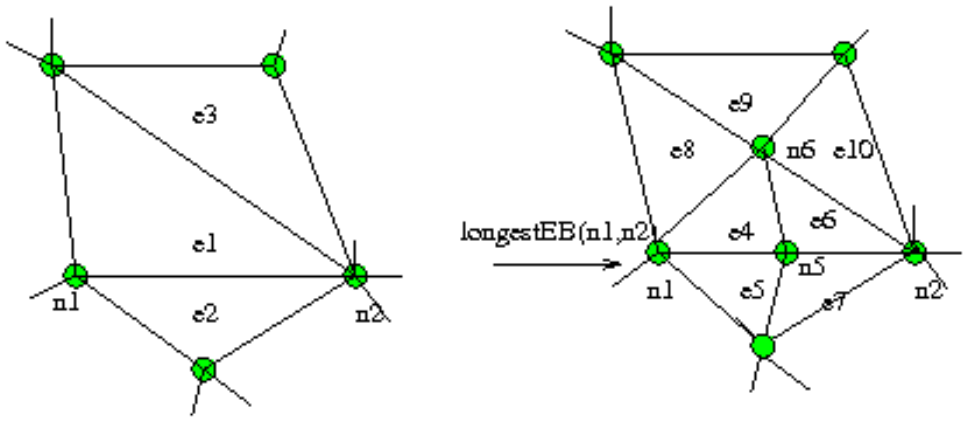

Figure 4.5: Longest Edge Bisect (n1,n2)

edge. Now, we realize that $(n1, n2)$ was not the longest edge on $e1$. So, we find the longest edge and bisect that edge. This is the description of the operation. The implementation is however slightly more complicated, and we will explain this in the following Chapter.

4.2.6 Vertex Remove

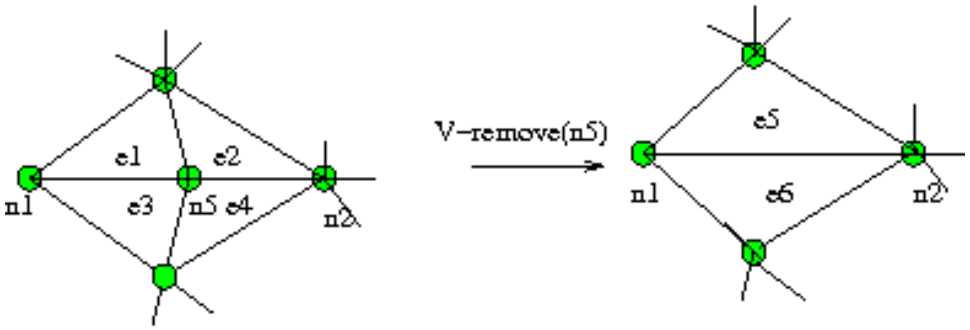

Figure 4.6: Vertex Remove (n5)

This is an operation which is the exact inverse of Edge Bisect. The figure 4.6 describes the operation. The implementation is also pretty straight-forward. The surrounding nodes are locked. Elements $e1$, $e2$, $e3$, $e4$ are removed. Node $n5$ is removed. Then $e5$ and $e6$ are added, and all the surrounding nodes are unlocked.

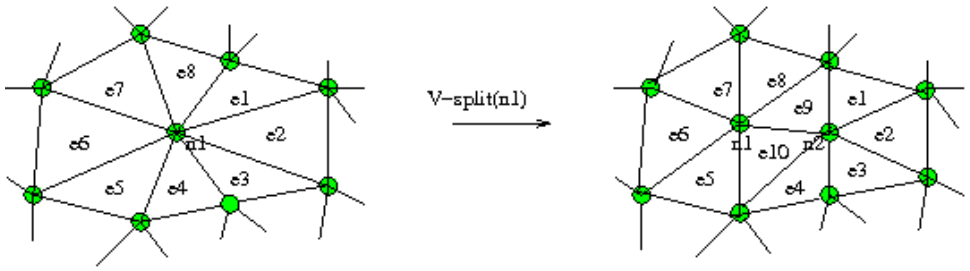

Figure 4.7: Vertex Split (n1)

4.2.7 Vertex Split

The Vertex Split operation is supposed to achieve the inverse effect of an Edge contract. It is supposed to generate two new elements $e9$ and $e10$, after forming a new node $n2$. The position of the nodes $n1$ and $n2$ is computed by creating an element of ideal quality. Finally, the connectivity of the surrounding elements are changed to be distributed among the two nodes. It also uses the standard locking, removing adding nodes, elements and finally unlocking all the surrounding nodes. This operation is not completely implemented yet.

All of these above operations are atomic and in some cases these operations while explained very simply here, are a lot more complicated because one has to consider the fact that any of the nodes or elements in an operation might be local or remote, so a lot of communication and synchronization between the chunks involved is required. Ghosts also need to be updated all the time. Besides, during Edge Coarsen, boundaries between chunk might need to move, as some elements are lost. We will give a feel of some of the problems involved in some of these operations in the following Chapter.

4.3 Mesh Adaptive Algorithms

Once the above atomic operations are available which work in parallel, the higher level adaptive algorithms can provide undivided attention to quality control. These algorithms can be oblivious of the parallel complexity in the lower layers. In fact, this lets them be able to apply some of the best available serial adaptive algorithms. Each chunk only thinks

about the elements it contains as the entire mesh and can apply adaptive operations to it. The operations on the edges on processor boundaries will be automatically carried over to the concerned chunks hidden by the lower layer operations.

Some of the mesh adaptive algorithms we use in this layer are:

- *Simple Edge refinement*: Every chunk sorts all the elements on it, and at any instant refines the edge with the longest edge length. This list of longest edge-length elements is maintained in a heap and is constantly updated. This provides a simple yet powerful refining algorithm that generates pretty good quality elements after the refining step.

- *Simple Edge Coarsening*: Every chunk sorts all the elements on it, based on the smallest edge-length elements and keeps collapsing the smallest edge, thus getting rid of elements that have worst aspect ratio and quality. This also uses a heap data structure and keeps updating it as operations are performed on the mesh.

- *Simple Mesh Smoothing*: This is a simple adaptive operation that most frameworks also provide. Every chunk can loop over all the elements and nodes it has and update them based on several criteria, without modifying the geometrical connectivity. It just uses the property that if we update the values on a shared node it needs to be propagated, which is easily performed by the lower layer.

- *Simple Mesh Repair*: Each chunk hunts down elements with bad quality and coarsens the shortest edge on these elements, thus getting rid of the bad quality elements, improving the overall quality of the mesh.

A lot of sophisticated quality metrics are being developed along with algorithms which use them to decide which elements to refine or coarsen. These are totally outside the scope of this thesis and will not be discussed.

4.4 Managing Ghost layers

ParFUM supports ghost layers which has been discussed in some detail in Section 3.3. However, ParFUM adaptivity at the moment has a limitation that it supports only ghosts which are 1-layer deep node ghosts.

As the geometry of any chunk is being modified, if any node that is involved in this is on the boundary, then there are some elements that are connected to that node and they are ghosts on all chunks that this node is shared on. Thus, any changes to these elements need to be broadcast to all the chunks that know about them.

The various conditions when ghosts need to be updated are:

- When a chunk is acquiring an element on another chunk, then in 2D, for triangular elements, two of the elements are shared, while the third node could either be shared, or it could be a ghost on the acquiring chunk. In the first case, when it is shared on the acquiring chunk, removing the element on the initial chunk and adding it to the acquiring chunk is a normal remove element and add element operation. However, when the node is a ghost, then it involves creating a new local node on the acquiring element, and copying all the solution attributes from the remote node. Apart from this, due to this new node being added, the acquiring chunk also needs to know about all elements connected to this node, local or remote. So, it needs to get this information from the initial chunk. As a result, new ghost nodes and ghost elements need to be added on this acquiring chunk. Also, the remote chunk might be losing a local node, which would imply, that it has to lose all ghost nodes and elements that was present on this chunk only because of that local node.

- When an edge on a chunk boundary is being contracted, a node is lost, and new ghost elements are added on both sides.

- When an edge with one node on a chunk boundary is being contracted, even in this

case, a new ghost node will be added, because the existing layer of ghost elements was deleted on the other chunks, and the new set of elements which fill up the space need to be updated on all chunks which need to know about this.

- When an edge on the chunk boundary is being refined, a new shared node needs to be added, and new ghosts need to added.

- When an edge with one node on the chunk boundary is refined, a new ghost is added and new ghost elements need to be added.

It is extremely important to bear in mind that the same operation on one chunk can be interpreted as a different situation on different chunks. e.g. some chunk can see it as a shared edge being modified, while others may see it as just one node being a shared node, and still others might be oblivious about the operation. In all cases, we would involve only the set of chunks that need to know about this modification, keeping minimal overhead and the final state after the operation should be consistent across all chunks.

Chapter 5

Parallel Adaptivity Implementation in ParFUM

ParFUM adaptivity has been implemented in a layered approach. The layers involved have already been described in much detail in the previous chapter. In this Chapter we will explain in more detail some of the data structures that have been used to implement parallel mesh modification, and also some other data structures that needed to be implemented to ensure proper operation in parallel.

We have already mentioned that ParFUM has been implemented in Charm++, particularly, Threaded Charm++. Here, each chunk has an individual thread of execution. The design inside adaptive calls however, is synchronous. All function calls made by one chunk (thread) on another is purely synchronous, hence there is synchronization involved at each remote function call. While this might not appear to be extremely conducive to the Charm++way of parallel programming, we can still get a lot of communication and computation overlap, by adding more threads on a processor. Thus, when a chunk(thread) waits for a synchronous function call to return, meanwhile other threads on that processor can use the compute power. In fact, in Section 6, we will note that the performance of the application improves by increasing the degree of virtualization, until the virtualization overhead takes over. One can completely kill the latency of synchronous calls through virtualization, which has been extensively used in the ParFUM adaptivity implementation.

5.1 Maintaining Adjacency Information

There are four important sets of adjacency information, which needs to be maintained at all times. In fact, these are maintained in a consistent state at the level of the parallel primitive operations. In fact, these are only updated during the add and remove element primitive operations. The other primitive operations do not modify these.

- *element to node(e2n)*: If an element is to be removed, this connectivity is deleted, while if an element is being added, this adjacency is updated by its connectivity.

- *element to element(e2e)*: If an element e is being removed, all adjacent elements have the entry for e in this adjacency list replaced by -1 (invalidated). When an element is added, the reverse is done. All elements that share an edge with this element will have this e added to this list.

- *node to element(n2e)*: If an element e is being removed, for all nodes that belong to its e2n, the entry for e is removed. If an element is being added, an entry for e is added to the n2n table.

- *node to node(n2n)*: If an element e is being deleted, only if an edge will be deleted, then for that edge $(n1, n2)$, the two nodes will lose their connectivity to each other. Similarly, when an element e is being added, if that creates a new edge between $(n1, n2)$, then both of the nodes will add the other added to their n2n.

The adjacencies are pretty much local information, and are defined even for ghosts, though they are treated slightly differently. For each chunk, the local mesh can be though of as all '*local* nodes and elements, and all ghost nodes and elements. Then the boundary of this can be thought of as the real boundary for this mesh. The adjacency of all nodes and elements (local or ghost) is consistent according to this abstract mesh.

5.2 Prioritized Locking

In Section 4, we have explained how locks are used to ensure atomic operations for mesh modification operations. In this section we will elaborate on locking. ParFUM uses prioritized node locking to implement the atomic operations and frees them.

Why lock nodes? We have the option to lock either nodes or elements. However, if we lock an element, then adjacent elements could start being modified, since it would be a separate lock. However, if a node is locked, no two different operations can modify different elements which include that node. Thus, locking a node provides us with an exclusive locking mechanism. So, we lock nodes instead of elements.

How to handle shared nodes? Shared nodes, mean there are multiple chunks with that node as a local element. To avoid a case where different chunks independently grab locks on different chunks, thus either incorrectly believing they have an exclusive lock or causing a deadlock, if they need to lock all the individual locks on each shared chunk. We designed the concept of a primary lock, where among all shared chunks, the chunk that has the smallest chunk id, is chosen to be the primary lock. Locking a node implies locking the primary lock. This decreases the total amount of communication needed for locking and also gets rid of deadlocks.

Are the locking operations blocking? Since a single operation needs multiple locks and there is no way to have a global ordering of all the nodes being locked, one can get into a deadlock, if blocking locks are implemented. Hence, we introduce the concept of non-blocking lock requests. If the lock is already acquired, the call request returns with a failure. Each chunk tries locking all nodes it needs a number of times. The locks that it has acquired are kept for some number of tries, and if all locks could not be acquired, then all locks are released, and the thread yields. It tries locking everything all over again. This theoretically gets rids of deadlocks, though there could be a substantial communication overhead.

Does that mean there could be live-locks? Technically, it is not possible if the higher level algorithm that uses it, learns which operations fail. Because it gives up trying an operation completely if it cannot get all locks after a random number of tries, which is different on all chunks and different at different times. However, operations might live-lock, if the higher level algorithm continuously issues a failed operation. However, with even the simplest of higher level algorithms and a huge number of chunks (a couple thousands) and really large meshes (billion elements), we do not get to live-locks. For all practical purposes, we can claim to be deadlock and live-lock free.

What is the priority? It is possible to suffer from a lot of messages if two chunks are competing for the same set of locks and each time both of them manage to grab some and not all. To improve a large number of these conditions, we keep a priority flag in the locks. If a lock is being requested by a number of chunks, the priority flag is set every time a chunk with a smaller id requests it, and the other chunks fail immediately, recognizing that someone with a higher priority is trying for that lock. Though it largely decreases the communication overhead, it does introduce a possibility of load-imbalance by giving priority to certain chunks. However, we expect to use Charm++ load-balancing capabilities to offset that.

5.3 Mesh Boundary

As we employ adaptive operations on the elements and nodes of a mesh, one could very easily destroy the shape of the mesh if not very careful. The framework needs to be careful about this. For this it obviously needs to be able to distinguish the boundary nodes from other nodes. Not just that, it also needs to be able to distinguish nodes on different boundaries, if it wants to coarsen edges on the boundary. Because, it needs to infer just from the boundary information if an edge connecting those nodes is on the boundary or local.

In figure 5.1, B1, B2 and B3 are three different physical boundaries on the mesh. Note

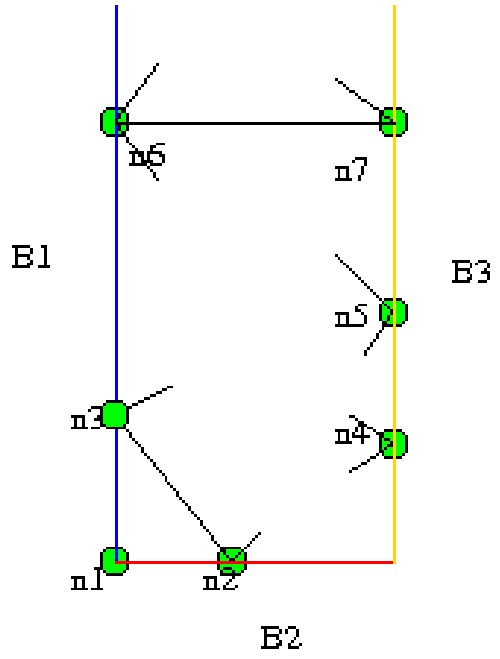

Figure 5.1: Mesh Boundary Idiosyncracies

that it should not be possible to contract the edges $(n3, n2)$ or $(n6, n7)$. The reason being that the two nodes involved are on different boundaries. However, one should be able to contract the edges $(n4, n5)$. Thus nodes on each boundary should have a different flag, to indicate that, otherwise the framework has no way to make out if an edge is on a physical boundary. Hence, all input meshes need to number each boundary with a different flag. We prefer the boundary number to be in certain order. We will talk more about this in the section 5.4

5.4 Mesh Corners

Corners define the physical shape of a mesh. This is a philosophy that ParFUM follows.The framework should never move corners. This also means that the set of corners is fixed for a mesh. No new corners could be added to the mesh. Hence, the set of corners for a mesh is computed first and this information is used forever. Refer to figure 5.1. What boundary flag should we assign for the node $n1$? At first glance one might say that is unimportant.

Consider what would happen if one tries to bisect the edge $(n1, n2)$. A new node $n8$ would be created between $n1$ and $n2$. How does one decide the boundary value for this new node. We could obviously solve that if we knew that $n1$ was a corner. So, the trick is to compute the set of corners initially, which could not be computed just based on the geometry, i.e. the connectivity of elements and nodes. For this we do need boundary information. Thus it is essential to have boundary information to compute corners. Now, if we assign corners arbitrary boundary flags, what would happen if $n1$ and $n2$, were both corners? What would be the boundary flag for $n8$?

One way to solve this would be to order the boundaries and choose the corners to have systematically either the higher or lower of the two adjacent boundaries, it is a corner on. This way we know what boundary flag to associate with $n8$, even when $n1$ and $n2$ are both corners.

5.5 Flexible Boundaries Between Chunks

ParFUM supports fluid chunk boundaries, meaning the chunk boundary can move to support various operations. In figure 5.2, nodes n2, n6, n4 and n2 form the boundary between C1 and C2. If Chunk C1 tries to perform EdgeContract(n4,n5), then elements $e4$ and $e5$ need to be deleted and the node $n4$ also needs to be deleted. However, it should be noted that if this is done, C1 still will believe that the node $n2$ is still local on it. It needs to realize that if it loses $e4$, the node $n2$ will have no local connectivity left and should be lost as a local node. Also C2 needs to realize that node $n5$ will now be a local node on C2. In fact this will be a shared node. So, while C1 should lose $n2$ and hence lose all ghost nodes and elements which it has only because of $n2$. At the same time, C2 should gain nodes $n5$ and all ghosts that should be present only because of $n5$. This sounds like a difficult operation.

The above set of operations can be simplified if C2 can acquire the element $e4$ at the beginning of the contract operation. This operation would exclusively take care of moving

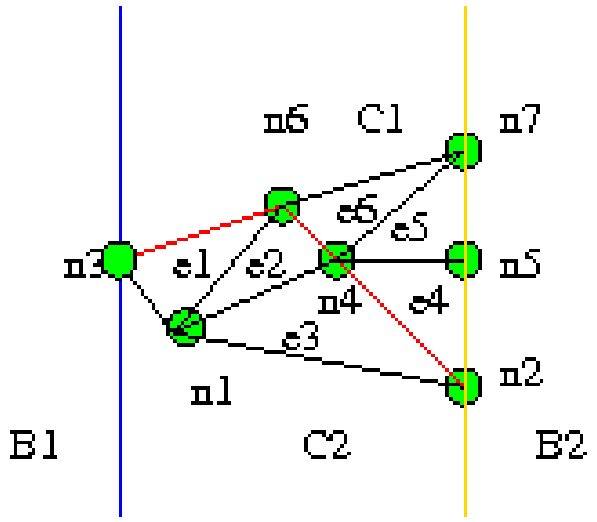

Figure 5.2: Need for fluid chunk boundaries

the boundary and adding and removing the new ghosts and shared nodes, thus, handling all IDXL lists for the concerned chunks. Now the EdgeContract operation can proceed normally between the two chunks.

5.6 Discontinuous Chunks

Because ParFUM allows fluid boundaries, there is nothing that ensures that the boundary of a chunk will remain continuous after a bout of adaptivity. Though the partitioner typically returns a continuous chunk, we are not sure if that is always guaranteed. Further, it is also possible to have a neighboring chunk acquire elements from another, such that it leaves the original chunk as discontinuous. In any way, it was imperative for us to support discontinuous chunks.

To support this, we needed to make sure that the implementation of none of the operations on a mesh/chunk, assumed anything about the connectivity of elements, i.e. local elements would be adjacent to at least one other local element, or some similar assumption which might seem logical on the face of it, but definitely breaks down under these special conditions. The design as an unstructured mesh to include all information in the adjacencies already ensured

that the framework could support discontinuous chunk partitions for a mesh.

However, this does result in a special case, which is also related to the next section. So we will describe it in detail in the section 5.7. What if one of the discontinuous sections of a chunk loses all its nodes and elements.

5.7 Empty Chunks

If there is only one element in a particular continuous region, and suddenly that element is also acquired by some other chunk, or this chunk wants to coarsen an edge on this element, thus effectively losing this element. How should this be handled?

This is one of the few special cases, when a chunk loses an element, it loses multiple local nodes and both chunks need to keep track of that. Apart from that after a chunk acquires a new element, it always has that element in the GER list of the losing chunk. However, here the losing chunk suddenly knows nothing about that element or any of the nodes on that element. This stumps the acquiring chunk, which needs to translate the indices back and tell the losing chunk, all ghosts that were created, because in this case none should be created.

Besides, even when a chunk loses all its nodes and elements, it still needs to participate in any global communication by the user, so it needs to be partly alive, though should not participate in any unnecessary communication during adaptivity.

5.8 Node from Local to Ghost

When an element $e1$ is acquired by chunk C2 from chunk C1 as shown in the figure 5.3, then it should be noted that this case is quite different from normal operation. In most cases, when a chunk loses a node, it becomes a shared node first, and is then deleted, Similarly, when a node is acquired, it moves from ghost to shared.

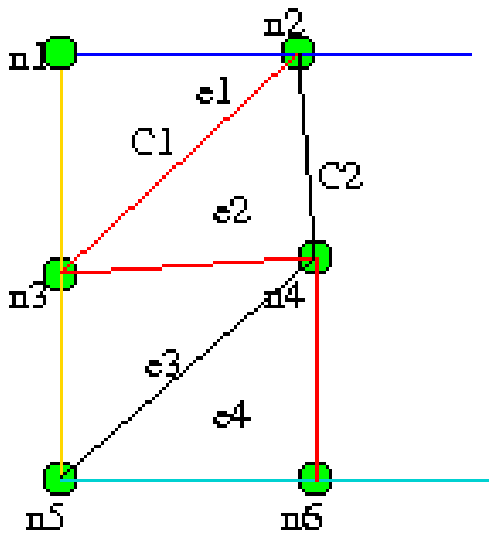

Figure 5.3: Transition from local to ghost node

This is unique because $n1$ was a local (not shared) node on C1, but after the operation it would be a ghost on C1. For C2, it transitions from a ghost node to a local node too. When a chunk acquires a node, the node is still shared on another chunk, so it can grab the solution data from that chunk. However, in this case $n1$ does not exist as a local on any chunk for a while, so its data needs to be transferred before it is deleted.

5.9 Solution Transfer

ParFUM entities, nodes and elements have solution data associated with them. When a new node is created, data needs to be interpolated on this new node. We provide a set of default interpolation operations for all user and system attributes. If the user so wishes, she can register a function pointer to be called during interpolation. The system attributes will still be interpolated in the default manner. The default operations the framework provides are:

- Interpolate the data based on an average between a set of nodes
- Copy the data from one entry to another for an entity.

For elements, the default interpolation function provided by the framework only copies from one entry in the entity to another. Sophisticated methods to distribute the data in the ratio of the areas, for the region that is being replaced have not been implemented yet, though according to the design it is possible.

5.10 Preserving Mesh Geometry correctness

ParFUM always ensures that none of its atomic operations results in an inconsistent mesh. However, this is for the unstructured domain. The framework needs to work extra hard, to ensure that it does not do something that might be valid in the unstructured domain, but makes no sense geometrically.

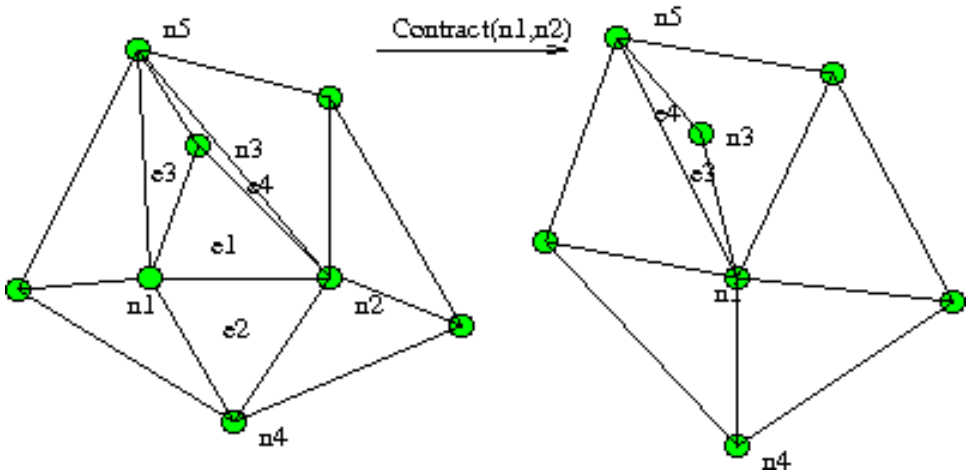

Figure 5.4: A Flip Destroying Mesh Geometry

An example condition is when the edge $(n1, n2)$ is contracted in the figure 5.4. After the flip, it should be noted that the elements $e3$ and $e4$ overlap each other, and in the geometry, definitely the connectivity of $n3$ seems weird (incomplete). This is because the element $e4$ actually flipped during the contract operation. The orientation of the nodes of $e4$ changed, causing an inconsistent geometry. This condition needs to be checked for during every EdgeContract operation.

The solution is to check the orientation of the nodes before and after the operation,

pretending the operation has been done, while not actually performing it, and in case, it causes a flip, not letting that operation through. Thus, we have tests which make sure that the generated geometry is correct.

In this section, we have only presented some of the important special cases and considerations to ensure correct parallel implementation. Apart from the above-mentioned, there are tons of other important cases:

- A chunk can suddenly have a shared or ghost nodes and elements with a completely new chunk as a result of an acquire operation by it, or some other chunk, or a edge contraction operation by some chunk.

- Similarly, a chunk can suddenly lose all IDXL list entries with another chunk.

- Because of an adaptive operation, a new ghost now comes from one chunk, however, one needs to verify if it was already a ghost from some other chunk and if it was, it needs to be mapped correctly and the old number should be properly added in the index lists.

It is not important to explain each of these, as we believe these are just too much detail that cannot be covered in the thesis.

Chapter 6

Parallel Adaptivity Results and Performance Analysis

ParFUM is designed to be used for high performance computing applications. Hence, it is extremely important to ensure the performance of the framework. In this thesis we are mostly concerned about the adaptive component of the framework. The computational performance of a high performance computing tool is as important as its memory utilization. So, this section is devoted to an analysis of the serial and parallel computational and memory performance of the adaptive component of the framework. The serial analysis is just a starting point to weed out any obvious performance bottlenecks for the parallel one.

6.1 Parallel Adaptivity Results

We present an example application here, which only performs adaptivity, and no physics, hence we can specifically study the adaptive component of ParFUM. We choose two geometries:

- A 750 element square. This mesh is regular, as it was generated by a simple script to generate different sizes meshes. This mesh is distributed on 6 processors.
- A 88 element L-shaped object, which is distributed among 3 processors.

For both the meshes, we initially perform 2 sets of alternative refinement and coarsening, keeping the target area for all elements the same. Then we start decreasing the target area to half for three consecutive refinement operations. Finally for five consecutive operations, we increase the target area for all elements on the mesh to twice on each step.

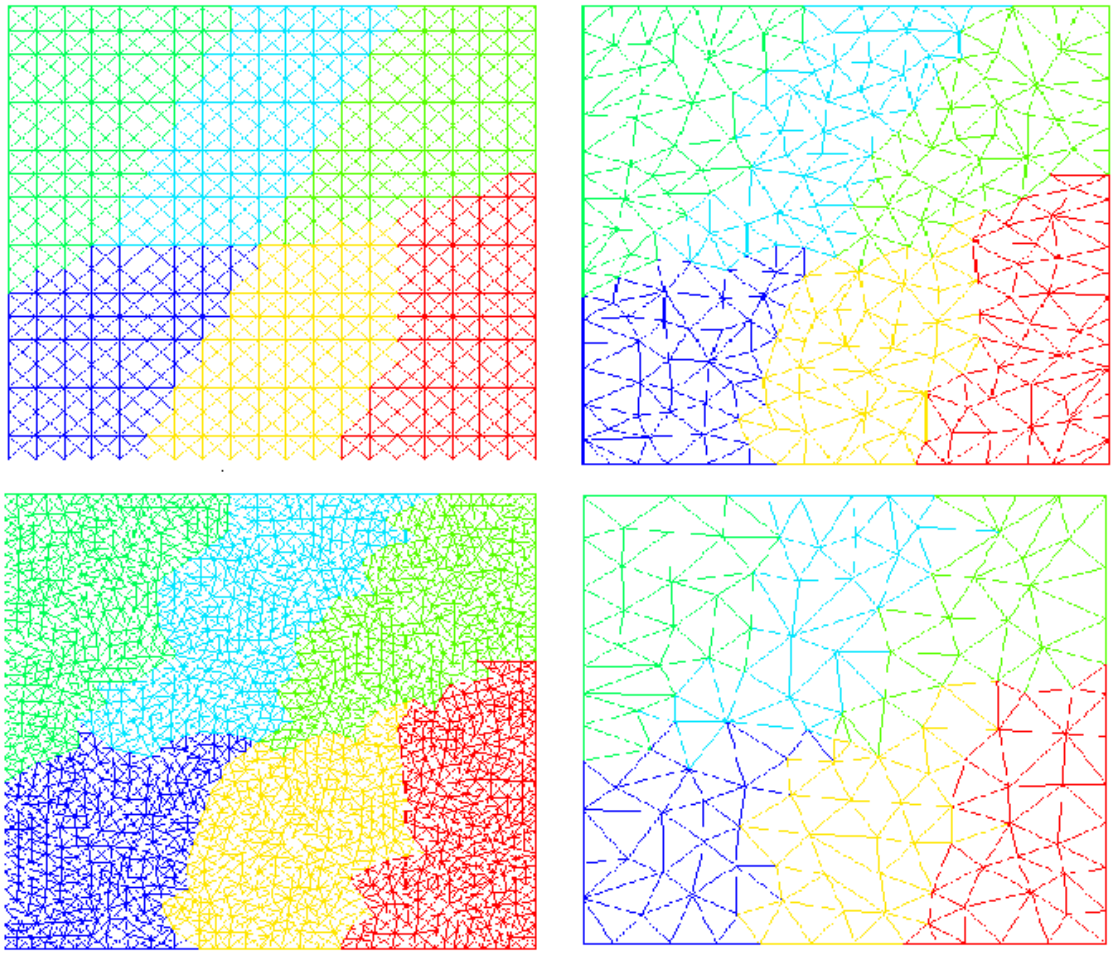

Figure 6.1: A set of adaptive operations on a 750-node square mesh on 6 processors

Figure 6.1 presents the geometry of the first mesh, initially, and after two sets of alternate refinements and coarsenings, then after three sets of refinements and finally after 5 sets of coarsenings.

Figure 6.2 shows a similar set of operations for the second geometry.

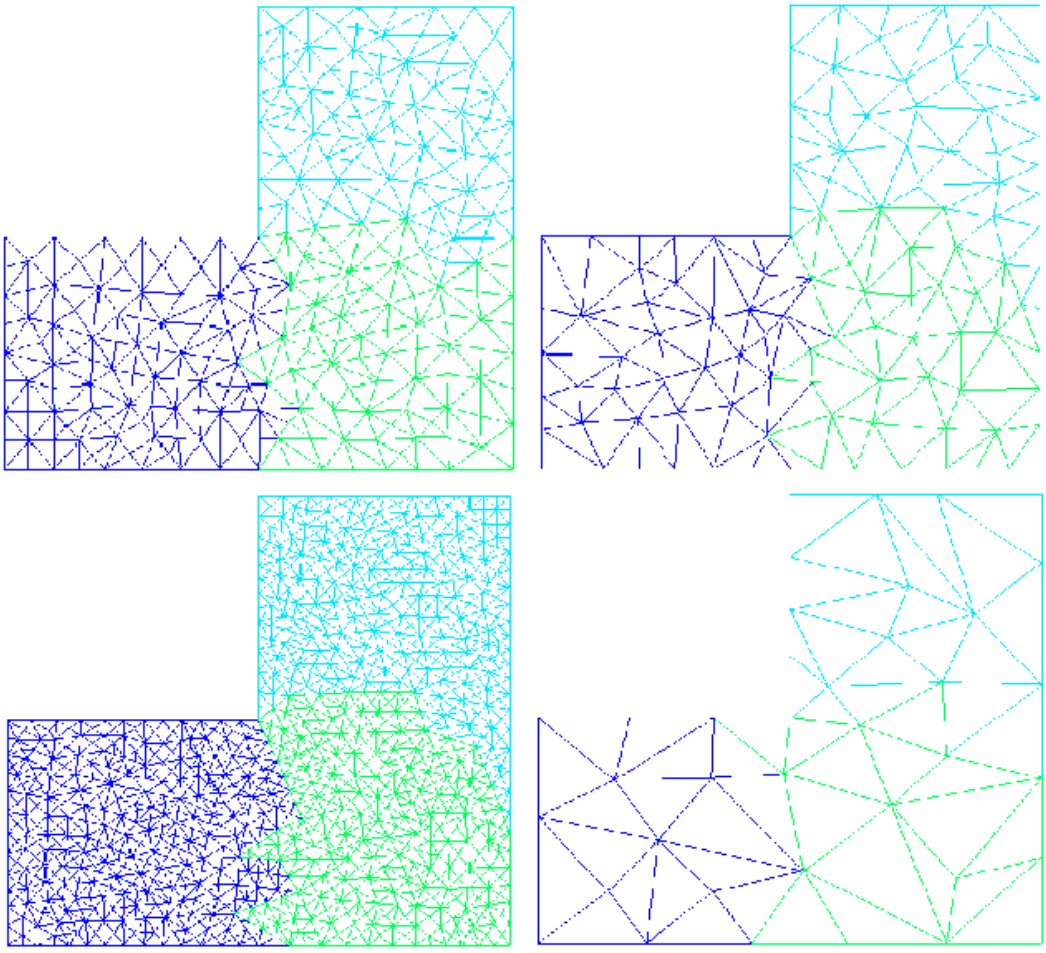

Figure 6.2: A set of adaptive operations on a 88-node L-shaped mesh on 3 processors

From the set of parallel runs that we formed, the following two subsections will explain two extremely important characteristics of the adaptivity code.

6.2 Parallel Adaptivity Performance Analysis

6.2.1 Benefits from Medium-Sized Chunks

A chunk is a virtual processor. It is a set of elements that is contained in one local mesh. Each chunk is a separate partition. We recognize that it is important to have at least a reasonable number of elements on each mesh. This is important because the smaller the

number of elements, the smaller the mesh and the higher the fraction of elements and nodes that will be on chunk boundaries, thus the ratio of communication to computation would be high. Besides, since most of the communication in adaptivity as we have mentioned in earlier sections is synchronous, so we waste some time over all this waiting.

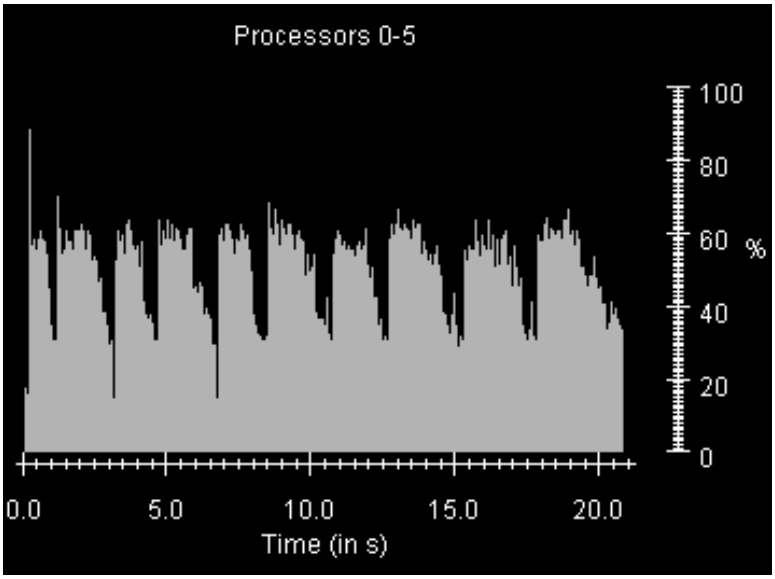

Figure 6.3: 10 sets of adaptive operations on a 13k element mesh across 6 processors

Figure 6.3 presents the utilization of processors over time, when a 13k element is refined and coarsened for 5 successive times across 6 processors. The average utilization across all processors over the entire duration of the program is less than 50%. During an adaptive operation, the performance reaches around 60% and stays there, but as the operation is about to end, and as some processors run out of work, the utilization decreases.

In the second example, there is a 240k element mesh spread across 12 processors. Only a single refine operation is performed in figure 6.4, while only a single coarsen step is performed in figure 6.5. The performance is consistently above 60%, but it drops of when some processors run out of work. This utilization drop can be minimized and the normal utilization could be improved if we use virtualization.

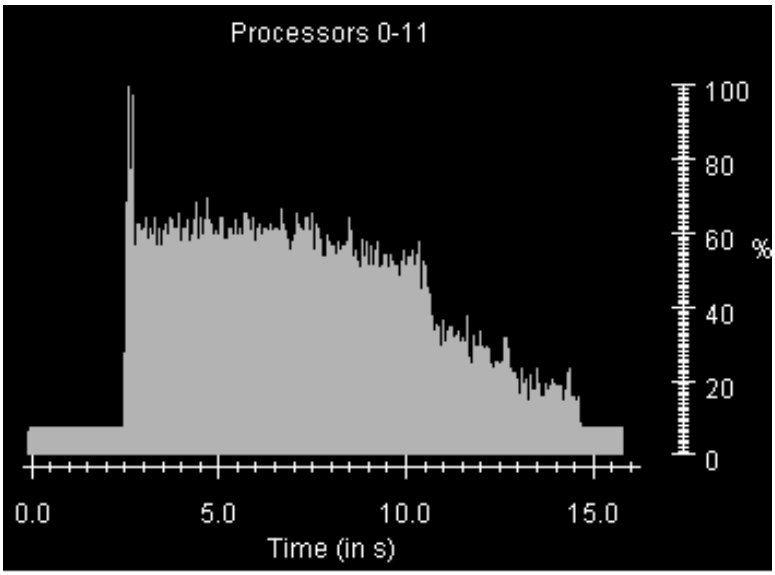

Figure 6.4: A single refinement operation on a 240k element mesh across 12 processors

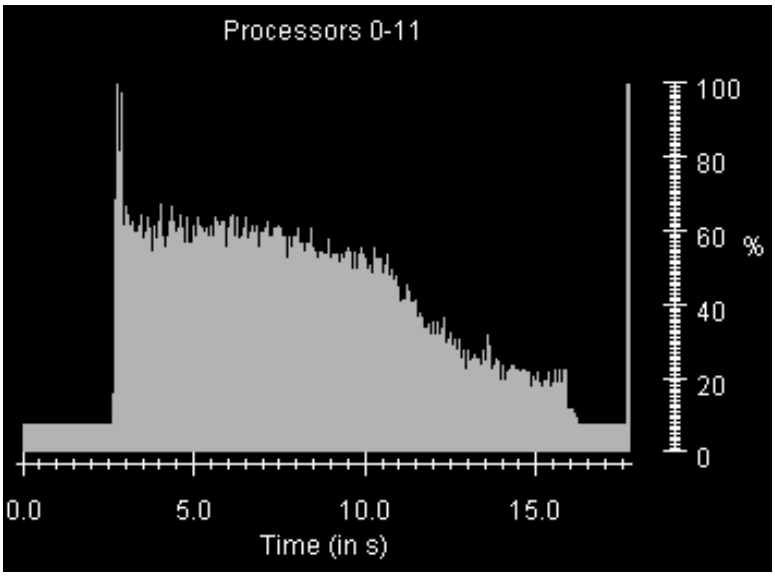

Figure 6.5: A single coarsening operation on a 240k element mesh across 12 processors

6.2.2 Benefits from Virtualization

In figure 6.6, we use a virtualization of 6 on each processor. This helps mask all the communication latency that we had seen in the earlier examples. Thus, we achieve almost 100% utilization most of the time, except the initialization when only one processor reads the input data, and finally when one of the processors runs out of work earlier. Actually this is nicely

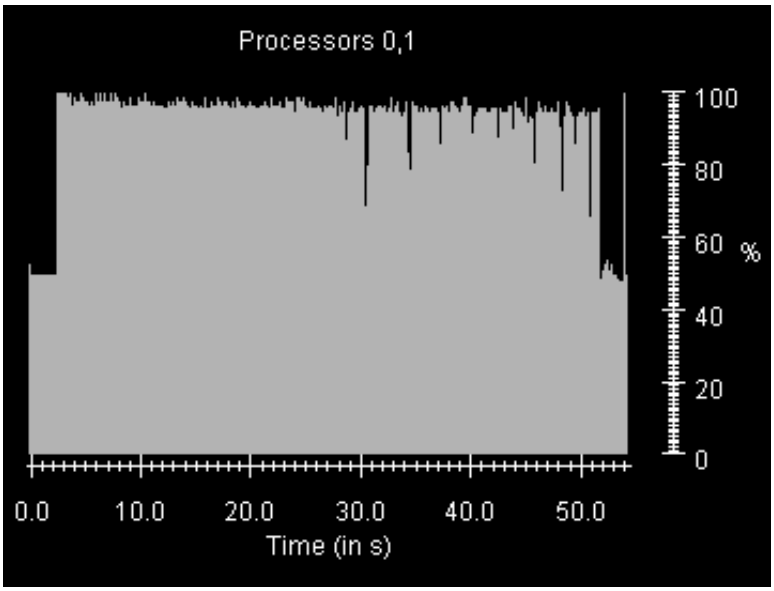

Figure 6.6: A single coarsening operation on a 240k element mesh across 12 chunks but 2 processors

tolerated, because some chunks on each of the processors start running out of work much earlier, and only when one chunk completely runs out of work, does the utilization drop, and this is not for too long, as the load had been nicely balanced by distributing virtual processors on the two physical processors. Thus virtualization provides important performance advantages for adaptivity.

Figure 6.7 presents the virtualization performance for a particular problem, introduced in the next section. We vary the number of chunks (virtualization) on 4 and 8 processors and plot the time taken for the same run. We note that initially with increasing virtualization, we get some performance benefits, as latency if tolerated by virtualization, but above a virtualization of 10, we start seeing the overhead of virtualization masking its benefits.

6.3 Parallel Performance Scaling

In section 7.2 we present an application that uses the adaptive framework. We also present some of its scaling characteristics. In this section we present two different problems that we

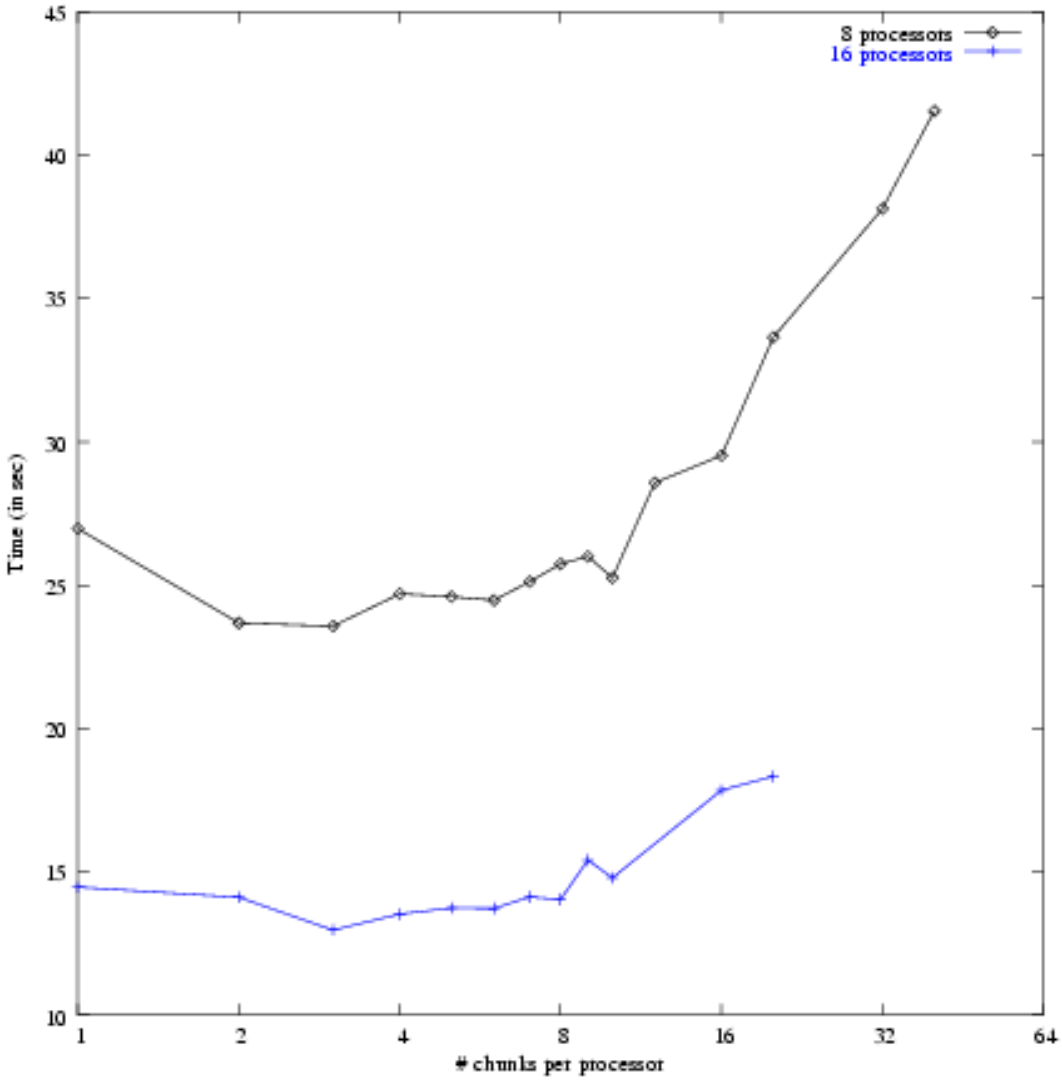

Figure 6.7: Demonstrate Virtualization benefits on Problem 1 for 8 and 16 processors

try to scale with the number of processors.

- *Problem 1*: A step of coarsening is applied to a 240,000 element input mesh and the resulting mesh has around 200,000 elements, so this involves coarsening 40,000 elements. This is not a large operation.

- *Problem 2*: A step of refinement is applied to a 240,000 element input mesh. The resulting mesh has around 300,000 elements. Then we coarsen this mesh in two steps.

The final mesh has around 200,000 elements.

We present scaling results from problem 1 and problem 2. Figure 6.8 presents the time

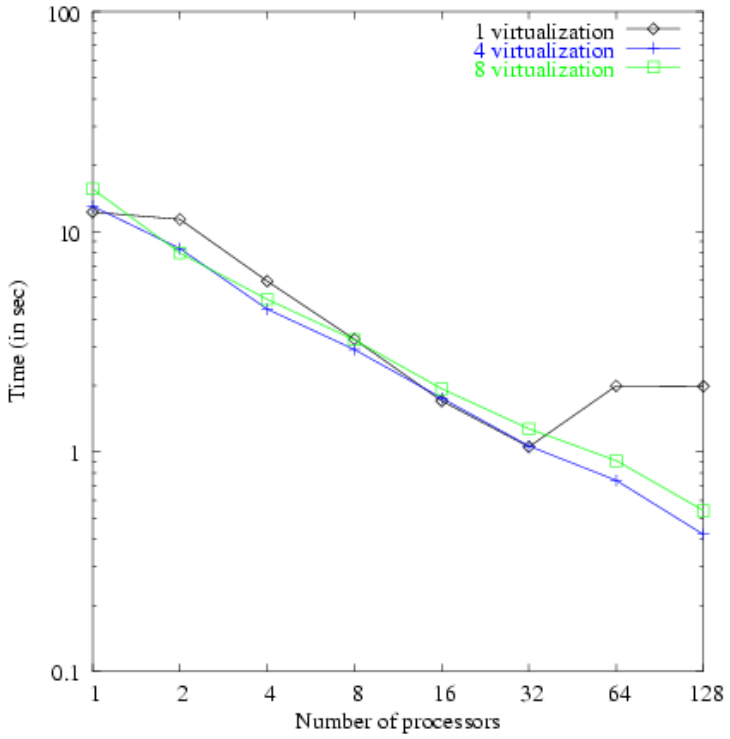

Figure 6.8: Speedup for Problem 1 at different virtualizations

taken when we increase the number of processors at each step by a factor of 2. There are three lines that denote different levels of virtualization. At a virtualization of 1, we note that there is not much benefit as we go from 1 to 2 processors. After this, it scales nicely to 32 processors, when the running time has become less than a second. Here there is not enough work, so breaking it up into more processors just increases the time required. Actually, this might be a special case, where a sudden lock contention caused the overall runtime to increase. Since the running time is small, one small abrupt behavior can change the total running time. For a virtuaization of 4 and 8, we have much better scaling throughout to 128 processors.

Figure 6.9 presents the same plot but for problem 2. We have a similar observation here. As we move from no chunks to 2 chunks, we take most of the performance hit. This is because

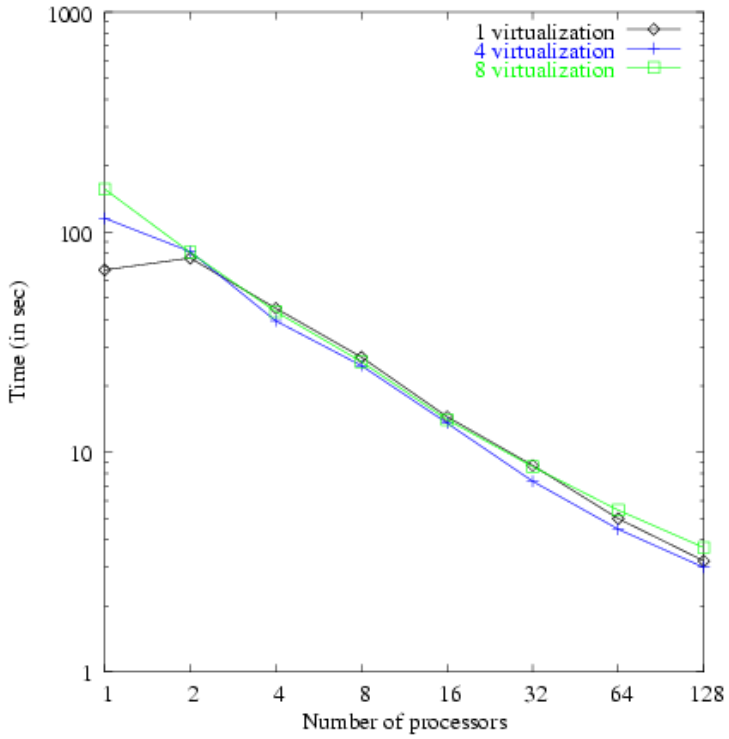

Figure 6.9: Speedup for Problem 2 at different virtualizations

none of the parallel stuff needed to be done when there was one chunk. Very importantly, at one chunk there was no ghost information, so this performance dip is completely acceptable, as the amount of work suddenly increased. The other two plots for virtualization 4 and 8 show a nice straight line, showing good uniform scaling.

In the previous set of graphs, we kept the virtualization degree constant and then increased the number of processors, thus the number of chunks increased for larger number of processors, clearly creating a disadvantage for them, as more chunks would inherently translate to more work. So, the scaling we saw was not ideal. The next two figures 6.10 and 6.11 try to look at the scaling characteristics from a different perspective. In these two graphs, we plot:

- *Best case performance*: The idea is to take the best case performance on different processors (We try a set of different virtualization and choose the best run time). This is the metric that the user really cares about. How much benefit would the user get by

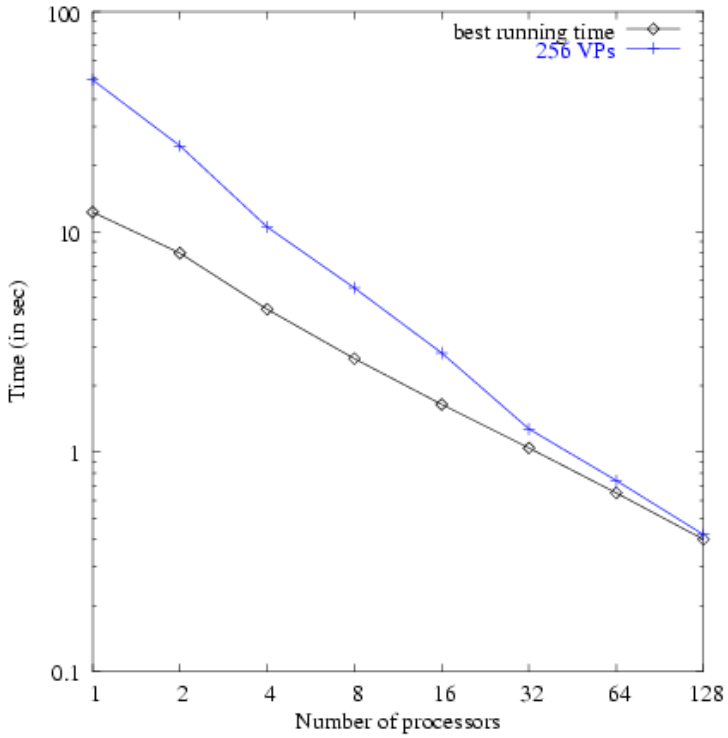

Figure 6.10: Speedup for Problem 1 at best performance and fixed number of VPs

adding more processors? The results here are biased as with more number of chunks, the amount of work increases.

- *Fixed number of chunks*: The idea is to have a fixed number of chunks (virtual processors) and run this on different number of actual processors. In our experiments, we fix the number of chunks to 256 and run it from 1 to 128 processors. The results here are also biased because a lot of chunks on smaller number of processors generates a large overhead, as we noticed in Figure 6.7.

Figure 6.10 shows the plot of time against number of processors for problem 1, while Figure 6.11 does that for problem 2. Both the measures of measuring scaling on both problems show uniform scaling. Only the magnitude of scaling varies largely according to the measure used.

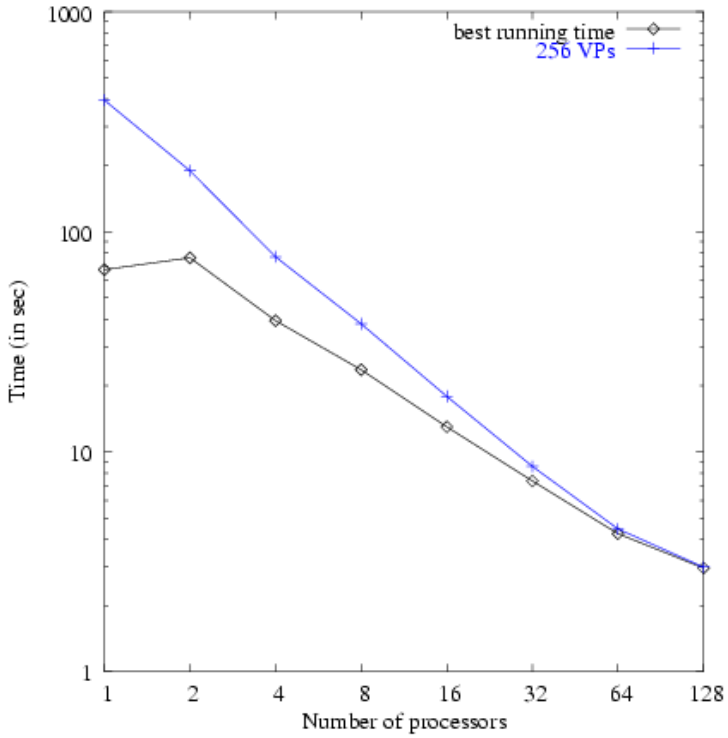

Figure 6.11: Speedup for Problem 2 at best performance and fixed number of VPs

Both these experiments demonstrate excellent scaling. The relative performance with respect to problem size and time of execution with increasing number of processors is pretty good to about 256 processors. We also have reason to believe that it will scale further, since we do not see any saturation in scaling. Moreover, as the communication characteristics of the adaptivity code does not blow up with increasing number of processors or chunks, we would expect good scaling.

There is however, no absolute metric to measure how the framework performs. So, all the performance analysis that we present is mostly relative. It would have been nice, if other such frameworks had existed that could perform geometric parallel adaptivity, which we could use to compare the performance of our design and implementation.

Operation	Percentage of Total Time Spent
Memory management (malloc / free)	35%
Maintaining and Updating IDXL	29%
Checking for Shared Nodes	16%
Updating element-to-element Adjacency	16%

Table 6.1: Profiled Information for Low Level Operations in Mesh Adaptivity

Operation	Percentage of Total Time Spent
Contract an edge	47%
Bisect an edge	33%
Add an element	37%
Remove an element	7%
Unlock Nodes	8%
Lock Nodes	5%

Table 6.2: Profiled Information for Mesh Adaptivity Abstract Functions

6.4 Profiling Parallel Adaptivity: Where does the Time go?

We use the standard GNU profiler to profile the performance of the application. The GNU profiler can only be used on a single processor. So, we could run the applications serially, or we could run it on multiple virtual processors but on the same physical processor. Finally, if run on actual physical processors, the final profiled file is generated only on one processor. However, this can give us enough information to identify bad performances and improve it.

Tables 6.1 and 6.2 present profiled information for the amount of time spent in different regions of the adaptivity code for an example program which only does mesh modification. It works on a 45k element square mesh and performs a series of 10 refinements and coarsenings alternatively. After the end of each refinement state there are twice the number of original elements, while the coarsening step leaves a new mesh with almost the same number of elements as the original. This runs on 16 virtual processors and 4 physical processors.

Table 6.1 shows that adaptivity spends a considerably large amount of time performing memory management. IDXL tables also use a lot of time and could be improved. Updating

element to element adjacencies also seems inefficient. All this collective information is used in the following section to perform optimizations to the code.

Table 6.2 presents a higher level view from the perspective of adaptivity, which are the locations it spends most of its time on. Contraction is a more expensive operation compared to bisection. This is also evident because we actually do less number of contractions that refinement in this example. Finally, since the locking is non-blocking, the locking code seems to be pretty efficient. Moreover, 'adding an element' is the most commonly used primitive among all lowest layer primitives. One should note that the higher level operations encompass some of the lower level operations, hence the times that are presented here are not mutually exclusive. It belongs to the entire operation and all its children.

6.5 CPU and Memory Performance Optimizations

- **LIFO list of invalid entries for entities**: The first set of results implied that the performance of the application largely depends on the size of the chunks that the application is working on. The performance characteristics of the application completely changes as the number of elements on a chunk changes. This clearly referred to the existence of some code which had a super-linear component. We soon realized that entities maintain only the first and last instance of invalid elements, so every search for a new invalid element means it has to iterate through the entire list until it comes across the next invalid element. If invalid elements are rare to find, this can take a really large time.

  To solve this problem, we decided to tradeoff memory to decrease the computation cost. A list of invalid elements is maintained in every entity. This list is a LIFO queue, where entries are added at the end, so it is a simple vector, but entries are only removed from the end. All we need to do is reuse entries, the order of reuse does not matter. So, the aim was just to minimize the overhead in the list maintenance, and it seems to

be least for a LIFO queue. It is $O(1)$ for insertion and deletion. It also does not waste any memory as the invalid entries list is always compact. This simple optimization improved the performance of the application described in section 7.2 by 100% i.e. it cut down the running time to half.

- **Memory Optimizations**: We try to decrease the number of mallocs by reusing any buffers we allocate as long as these buffers are of the same size. This very simple memory optimization, decreased the number of calls to malloc by a large fraction, but the improvements in time were not proportional to the decrease in the number of calls to malloc. However, there was still a significant improvement in the performance of applications.

- **Optimizing IDXL lists to better support adaptivity**: This still needs to be done. Firstly, IDXL entries need to be reused. The next optimization involves managing the reverse lookup data-structure more efficiently, so that instead of being reconstructed on every modification to the IDXL lists, they are dynamically modified incrementally as new entries are added or deleted.

Chapter 7

Applications using Adaptivity

Though ParFUM has been used and is being used for a large number of applications, this thesis, only focuses on applications that have used the adaptivity available in ParFUM. Apart from the fact that ParFUM greatly simplified the parallel interface for all these applications, the adaptive component has provided widespread advantages to each of these applications.

7.1 1D Wave Propagation Problem

For an ongoing research collaboration with Dr. Geubelle, we have used the mesh adaptivity in ParFUM. In solving a 1-d wave propagation problem on a bar, we wish to capture information at the shock front. One end of a 2-d bar is fixed and the other end is pulled with constant velocity as shown in figure 7.1. This results in a wave front that travels from the pulled end. To best capture the shock-wave front a very fine mesh is desired. A coarse mesh is less accurate. However, starting with the entire mesh very refined is very slow, so adaptivity is used to refine the mesh at the moving wave front and maintain a coarsened mesh elsewhere. A normalized velocity gradient is used as the criteria to identify the shock and modify the mesh size accordingly.

Preliminary results indicate that the regionally refined and coarsened mesh exhibits similar accuracy to an initially fine mesh for modelling the shock-wave front. Since the region of refinement is constantly changing in this simulation, it is also a challenging problem for

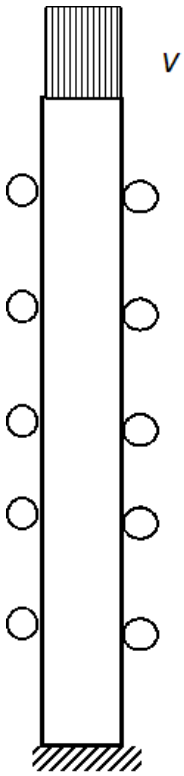

Figure 7.1: Shock-wave on a bar

Charm++'s load balancing algorithms to tackle.

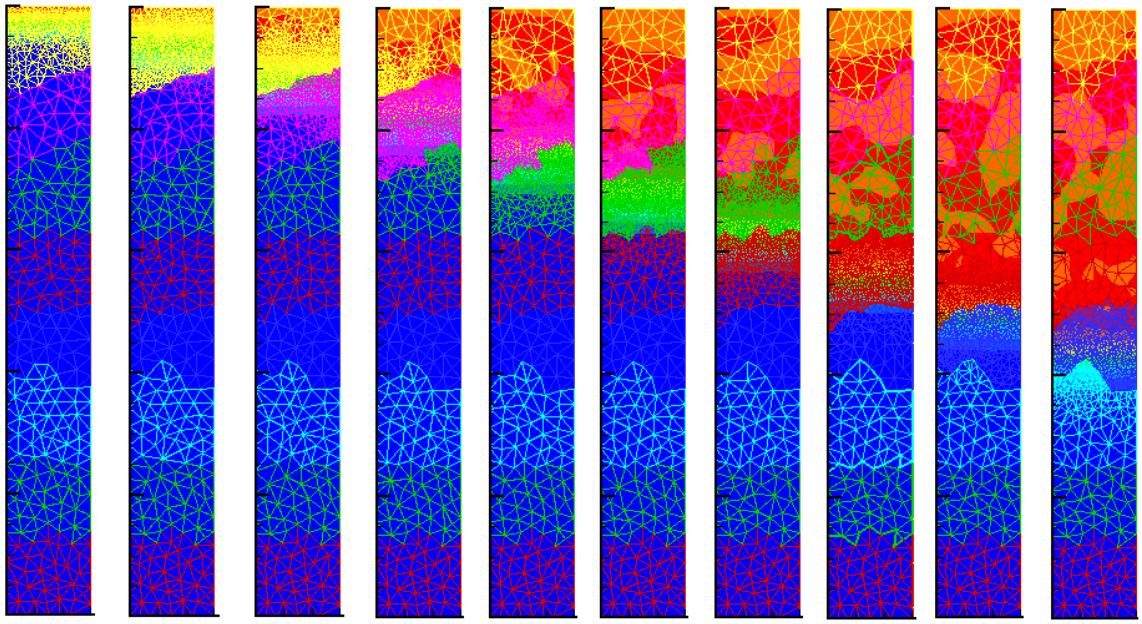

Figure 7.2: Mesh and velocity for Shock-wave on a bar on 8 processors

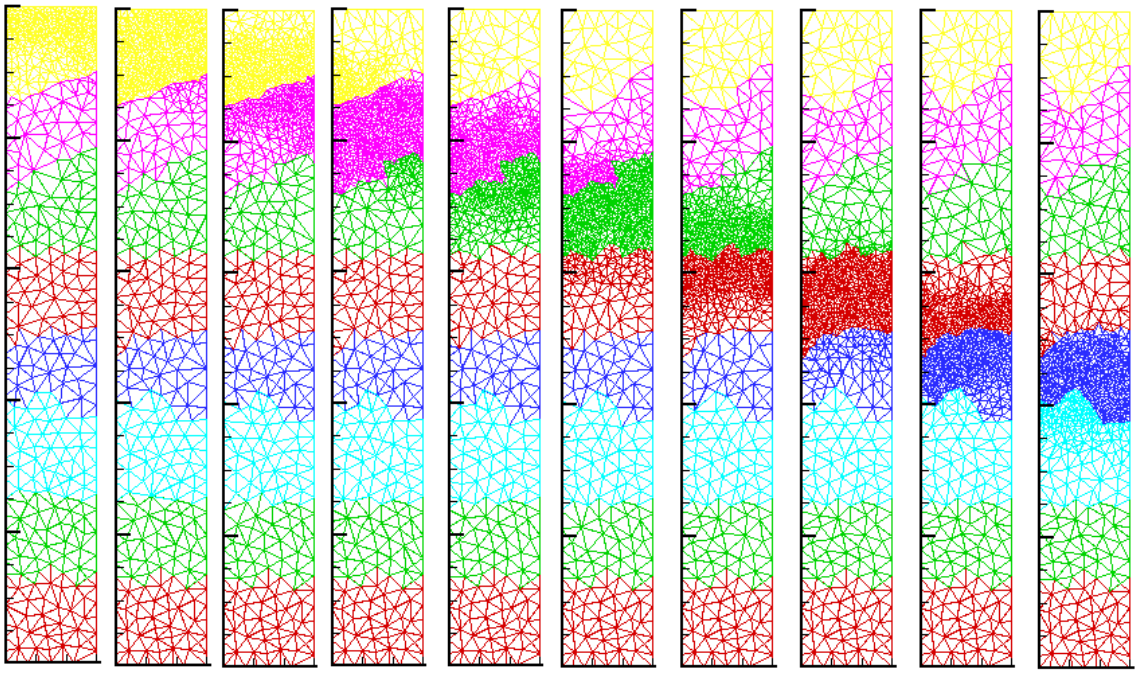

Figure 7.3: Mesh for Shock-wave on a bar on 8 processors

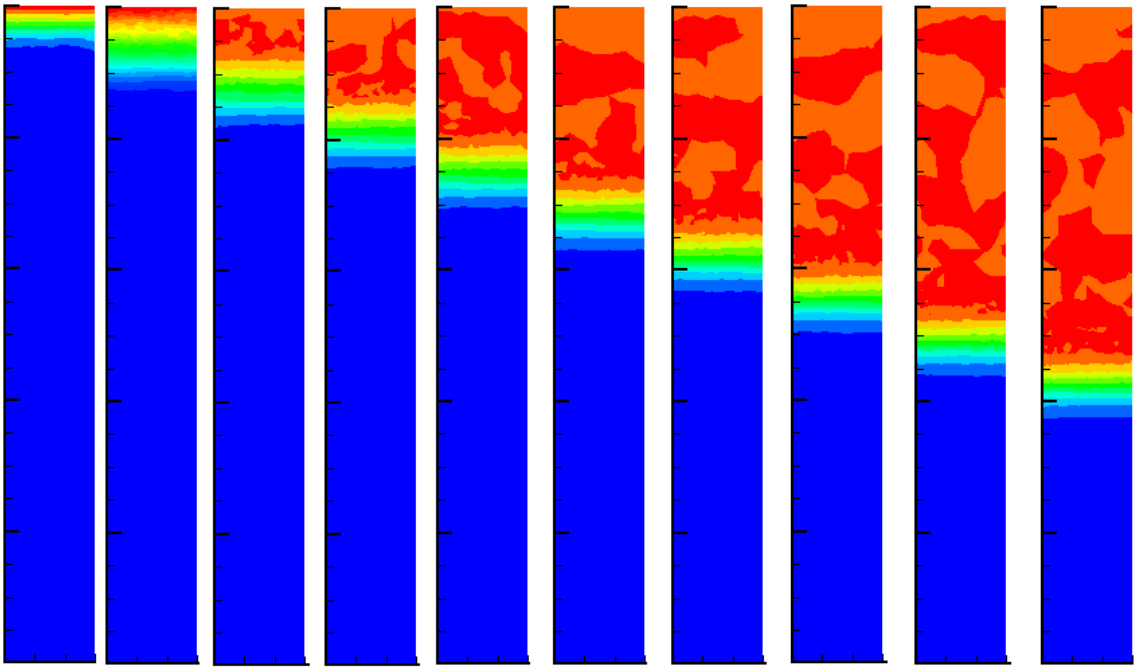

Figure 7.4: Velocity for Shock-wave on a bar on 8 processors

Figure 7.3 shows the Mesh divided on 8 processors and as the wave propagates, the region of refinement moves seamlessly across processor boundaries, coarsening previously

refined areas.

Figure 7.4 shows the solution data, which is the velocity at each point on the bar, which is actually computed on each of the elements on the bar. The solution is completely smooth, and works perfectly oblivious of the actual position of that element.

Finally, Figure 7.2 shows both the mesh partitions and the velocity solution data on the same picture, to bring the entire thing in perspective.

7.2 Generating a Billion Element Mesh by Continuous Refinement

This is a simple application, which does no physics. It only exploits the adaptivity in ParFUM to generate a huge mesh. In fact as the name indicates, the target was to generate a mesh with a billion elements. This is indeed a challenging problem for the adaptive framework. Not only does parallel mesh adaptivity need to work correctly, but it should also work with no memory leaks and good performance. And of course it should scale to a large number of processors. The first step was to determine the maximum sized initial mesh that the parallel partitioner in ParFUM could partition easily to a few thousand chunks, so that we could run this on a few hundred processors.

After a number of trials, we honed on the optimal and largest partition that the parallel partitioner can perform, a one million element initial mesh, was partitioned into 2048 chunks on 256 processors. This meant, each chunk had on an average of 480 elements to start with. To reach one billion elements, we would need about 500,000 elements on each chunk finally. We decided to refine it to half the target size in each iteration. This would require approximately 10 iterations of refinements to reach a billion element mesh. This experiment was run on Turing [1]. We used both processors on each node, thus using a combined memory of half a terabyte of memory. It took a little above 3 hours to complete 10 iterations and

[1] *Turing* is a cluster of 640 dual 2GHz G5 processors, 4GB Apple Xserves connected by Myrinet

reach to a little over a billion elements.

A smaller run had shown that on 64 of the same processors, it generates 250 million elements, but runs out of memory before it reaches 500 million elements. Thus, on 256 processors, we have a memory limitation to reach to the next step of the iteration. This experiment was run with a virtualization ratio of 8. A higher virtualization ratio, somewhere around 10 or 12 would have been preferred, but we could not do this because we could not partition the initial mesh into a large number of chunks. The dilemma here was to get a reasonable balance between chunk size and the number of chunks. We wanted a larger number of chunks as well as more elements on each chunk, so that the number of ghosts is not a very large fraction of the number of actual elements. This translated to a large mesh, to be partitioned into a large number of chunks. Because of the memory limitation, we plan to use a single processor per node so that it can have twice the amount of memory. However, the partitioner could not partition it on 512 processors. We are looking at alternative ways to run this on 512 processors with the same virtualization ratio.

Figure 7.5 plots the total number of elements per chunk on the X axis. The Y axis denotes the time taken by each run. Since the virtualization is fixed at 8, the total number of elements per processor is same across all the runs. There are 4 runs, one each for 32, 64, 128 and 256 processors. This graph tries to show that the problem size scales perfectly with the number of processors. If one increases the number of processors by a factor of 2, it can solve a problem of twice the size in the same time. All the runs follow the same line on the graph, demonstrating that each iteration of adaptivity scales perfectly to 256 processors. We did not get access to more processors, and the partitioner is also a limitation. We are pretty confident that it will scale beyond this. This is primarily because most communication is limited between the chunks that share a region in geometry. So, the neighboring set of processors which communicate is usually a small set. As communication per chunk is independent of the number of chunks largely, unless the number of chunks is too high or too low, we see pretty good scaling.

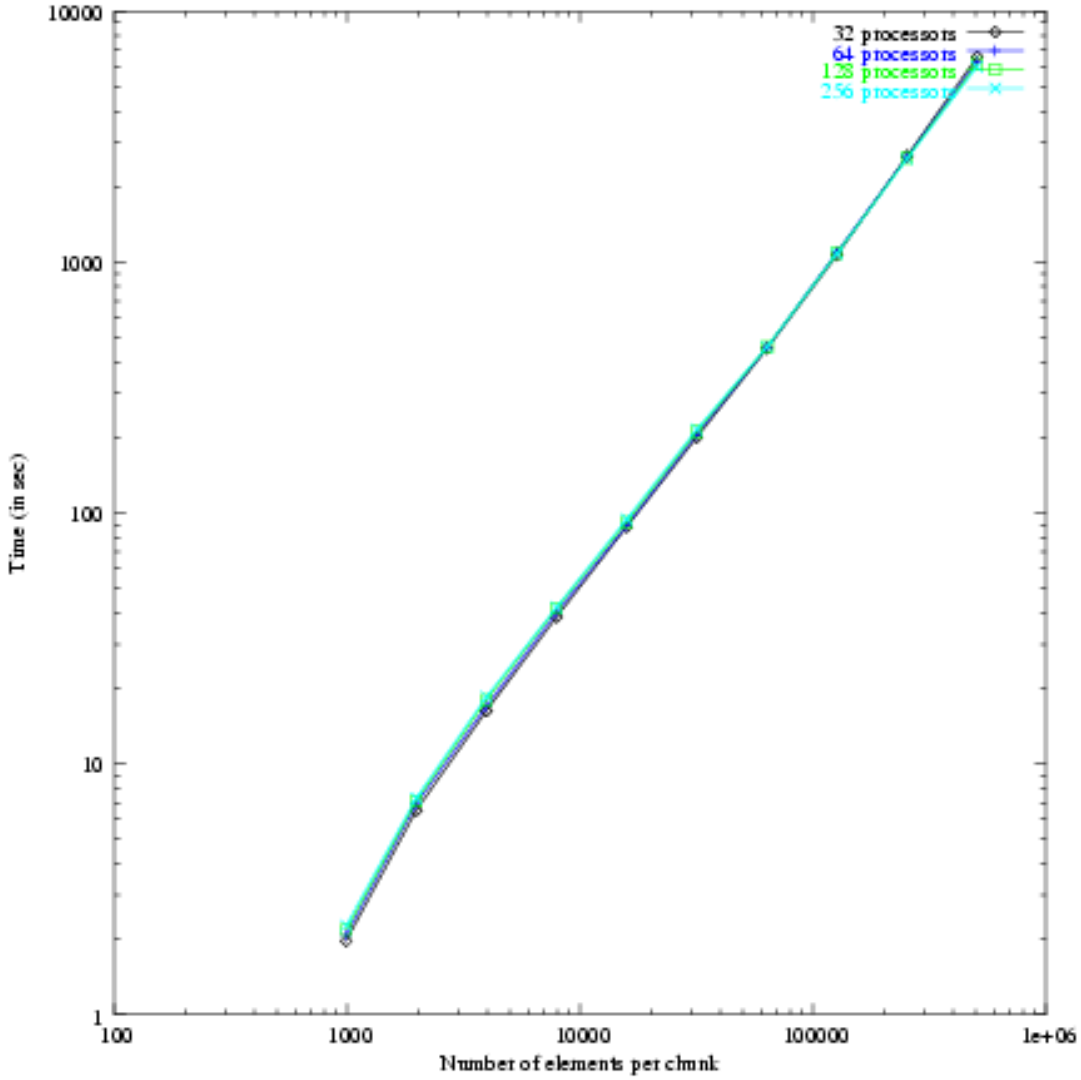

Figure 7.5: Scaling problem size/processor with time at virtualization of 8

Apart from good scaling in the problem size domain, we also note that even in the time domain, when the problem size for each run doubles on the same set of processors, the time for the operation increases slightly more than double. Things do not scale perfectly in this sense, but it is close to ideal scaling and most importantly we have a straight line scaling till we finally run out of memory to accommodate the mesh. At the time it runs out of memory, each processor has around 6 million elements.

7.3 Spacetime Discontinuous Galerkin

The Spacetime Discontinuous Galerkin (SDG) method [1, 9] provides a powerful way to analyze phenomena such as shock wave propagation in solids, evolution equations for state variables in inelastic constitutive models and Hamilton-Jacobi level set models for interface kinetics.

The SDG method uses discrete basis functions in space and time over partitions of the spacetime domain. The current implementation of the SDG method employs unstructured spacetime meshes satisfying a special causality constraint. For such meshes, the SDG method can be implemented as an advancing front solution technique which interleaves the generation of a patch of a small number of elements and the solution procedure in the patch. The space domain to be analyzed is represented by an unstructured mesh referred to as the space mesh.

In the adaptive version, the space mesh is refined or coarsened to obtain a more accurate solution at points of interest without paying a high cost all over the domain. The non-adaptive version does not change the space mesh.

This is still a work in progress and we are working on using the adaptive version of ParFUM for this application.

Chapter 8

Conclusion and Future Work

This thesis contributes a parallel, fully adaptive, asynchronous, incremental, mesh adaptivity component to ParFUM. Any application that uses ParFUM can use the adaptive component to dynamically refine and coarsen regions of the mesh it is working on. This could be just to improve the quality of a region, or to solve the physics in a region in much greater detail than in other regions. Since the adaptive component is built on top of Charm++ / AMPI, it supports automatic overlap of computation and communication.

The adaptive component of ParFUM makes the feature list that ParFUM provides extremely impressive, and we hope more and more applications will take lead from the few applications that have already benefited from the adaptivity component and will use the immense benefits that it provides.

More sophisticated quality conscious metrics and sophisticated refinement, coarsening, smoothing and repair algorithms are in the process of being developed which will utilize the adaptive operations that ParFUM provides in a much better manner, allowing applications to do much more with mesh adaptivity.

Load Balancing in the adaptivity code is still being implemented and we would be able to solve many more problems much more efficiently soon.

Parallel Performance of the mesh adaptivity feature is still undergoing some improvements. We have identified and fixed some bottlenecks to performance, while some are yet to be completely removed, though they have been identified. As we continue to improve

the framework and its performance, we expect more application scientists and engineers to derive its benefits. We sincerely hope that this provides scientists with a tool to overcome existing computational challenges and solve interesting problems.

References

[1] Reza Abedi, Shuo-Heng Chung, Jeff Erickson, Yong Fan, Michael Garland, Damrong Guoy, Robert Haber, John M. Sullivan, Shripad Thite, and Yuan Zhou. Spacetime meshing with adaptive refinement and coarsening. In *SCG '04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 300–309, New York, NY, USA, 2004. ACM Press.

[2] Kevin Barker, Andrey N. Chernikov, Nikos Chrisochoides, and Keshav Pingali. A load balancing framework for adaptive and asynchronous applications. *Institute of Electrical and Electronics Engineers Transactions of Parallel and Distributed Systems*, 15(2):183–192, 2004.

[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, pages 385–395. Springer Verlag, December 2000.

[4] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[5] H. L. DeCougny and M. S. Shephard. Parallel refinement and coarsening of tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 46(7):1101–1125, 1999.

[6] Jayant DeSouza and Laxmikant V. Kalé. MSA: Multiphase specifically shared arrays. In *Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, United States of America, September 2004.

[7] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[8] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, pages 306–322, College Station, Texas, October 2003.

[9] L. V. Kale, Robert Haber, Jonathan Booth, Shripad Thite, and Jayandran Palaniappan. An efficient parallel implementation of the spacetime discontinuous galerkin method using charm++. In *Proceedings of the 4th Symposium on Trends in Unstructured Mesh Generation at the 7th US National Congress on Computational Mechanics*, 2003.

[10] L. V. Kale 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.

[11] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at International Symposium on High Performance Computer Architecture)*, Madrid, Spain, February 2004.

[12] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, pages 23–32, Melbourne, Australia, June 2003.

[13] Orion Lawlor, Sayantan Chakravorty, Terry Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, (to appear), 2006.

[14] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[15] James R. Stewart and H. Carter Edwards. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elements in Analysis and Design*, 40:1599–1617, 2004.