A Hybrid Parallel Framework for Computational Solid Mechanics

by

Piotr Fidkowski

S.B., Civil Engineering, Massachusetts Institute of Technology (2009)
S.B., Aeronautics and Astronautics, Massachusetts Institute of Technology (2009)

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

Author . . . .
Department of Aeronautics and Astronautics
May 19, 2011

Certified by . . . .
Raúl A. Radovitzky
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by . . . .
Eytan H. Modiano
Associate Professor of Aeronautics and Astronautics
Chair, Graduate Program Committee

A Hybrid Parallel Framework for Computational Solid Mechanics

by

Piotr Fidkowski

Submitted to the Department of Aeronautics and Astronautics
on May 19, 2011, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

A novel, hybrid parallel C++ framework for computational solid mechanics is developed and presented. The modular and extensible design of this framework allows it to support a wide variety of numerical schemes including discontinuous Galerkin formulations and higher order methods, multiphysics problems, hybrid meshes made of different types of elements and a number of different linear and non-linear solvers. In addition, native, seamless support is included for hardware acceleration by Graphics Processing Units (GPUs) via NVIDIA's CUDA architecture for both single GPU workstations and heterogenous clusters of GPUs. The capabilities of the framework are demonstrated through a series of sample problems, including a laser induced cylindrical shock propagation, a dynamic problem involving a micro-truss array made of millions of elements, and a tension problem involving a shape memory alloy with a multifield formulation to model the superelastic effect.

Thesis Supervisor: Raúl A. Radovitzky
Title: Associate Professor of Aeronautics and Astronautics

Acknowledgments

I would like to recognize and thank all the people that contributed ideas and support to help make this work possible. I thank my advisor, Professor Radovitzky for his advice and direction. Many of the designs in this work were born out of lively white-board discussions with my colleagues, and I would like to thank Julian Rimoli for countless contributions, as well as the members of my research group, Michelle Nyein, Lei Qiao, Andrew Seagraves, Brandon Talamini, and Mike Tupek. I must especially thank Lei Qiao for developing the superelastic material model and aiding me in understanding and implementing it. The GPU work presented here was done at Argonne National Lab, and would not have been possible without the aid of my mentors Pavan Balaji and Jeff Hammond.

I wish to acknowledge the support of my parents and brothers, as well as the support of the friends who made my house in Cambridge a second home. I am additionally thankful to my friends throughout Boston and beyond for providing plenty of diversions outside of research.

This work was made possible in part by the Computational Science Graduate Fellowship provided by the U.S. Department of Energy and administered by the Krell Institute, under grant number DE-FG02-97ER25308.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, computational science has become one of the primary pillars of scientific research, alongside theory and experimentation. A number of different techniques and technologies have been developed to advance the state of the art in simulation. In the particular case of the finite element method, many computer codes were developed in Fortran in the 1970s to solve the continuous Galerkin finite element problem. However, many of these codes were written to solve one specific application, and few are flexible enough to handle the newest schemes and technological improvements such as GPU parallelization [1]. These limitations of code structure make it difficult to integrate the latest research that is required to solve ever larger and more complicated problems.

Computationally understanding complex material phenomena such as fracture, plasticity, and impact requires a combination of new numerical methods and improved computing power. Discontinuous Galerkin formulations have recently shown promise in brittle fracture problems [2], and also have beneficial conservation properties for wave propagation. The desire to accurately and efficiently capture wave interaction in impact and fracture problems motivates the need for higher order elements. Solving thermoelastic problems or problems involving superelastic materials can require coupled, multi-physics approaches with multiple unknown fields and equation sets on possibly different interpolations. Introducing length scales in plasticity can be accomplished through the use of strain gradients, which have higher order continuity requirements and may involve coupled systems of PDEs. Pushing the state of the art in computational solid mechanics will require us to take advantage of all of these numerical schemes, and will require new algorithms and data structures to handle the interactions.

Beyond these numerical technologies, we also need increased computational power to properly resolve the length and time scales in our problems. Modeling intricate geometries such as a human brain for blast problems requires a large number of degrees of freedom. Resolving the proper length scales in plasticity problems can require very refined meshes. Many modern finite element codes parallelize over distributed memory clusters to allow for solution on larger meshes or to reduce computation time. Recently, however, shared memory multicore technology such as Graphics Processing Units (GPUs) have gained prominence due to their impressive arithmetic ability and high power efficiency. Taking advantage of the potential speed increases offered by these new technologies requires new parallel algorithms. We ultimately envision a simulation capability where the limit in problem size is given by the available hardware and not by the scalability of the software architecture.

The main purpose of this thesis is to address this need and opportunity in computational solid mechanics and develop an extensible, object-oriented finite element framework with the goal of efficiently supporting new technologies and being modular enough to be extensible to unanticipated new methods.

1.2 Background

1.2.1 Finite element method

Ultimately, any computer code for the finite element method is a translation of the mathematical formalism in the numerical scheme to a computer program. Different mathematical concepts have implementations as various functions, variables and classes. As a reference for our later discussion of finite element frameworks, we will briefly review the overall mathematical structure for a static problem. Further details can be found in numerous references, including [3, 4].

We would like a numerical method to solve the general variational formulation

$$a(\bar{u} + u^g, v) = l(v), \qquad \forall v \in \mathcal{V}, \tag{1.1}$$

where $\mathcal{V}$ is a suitable space of functions over our domain Ω that vanish on the Dirichlet portion of $\partial\Omega$. The solution is separated into a function u^g that satisfies the Dirichlet boundary conditions and $\bar{u} \in \mathcal{V}$. We also assume that the forms a and l are linear in v.

1. Construct a discrete approximation Ω_h to the original problem domain Ω.

2. Determine a finite element triangulation $\mathcal{T}_h$ for the domain Ω_h, and for each element $K \in \mathcal{T}_h$ establish a map φ_K from the reference element to the element domain in Ω_h.

3. Choose a function space on each $K \in \mathcal{T}_h$, thus establishing an approximate function space $\mathcal{V}_h$ to the original function space $\mathcal{V}(\Omega_h)$. Given N degrees of freedom, we have that $\mathcal{V}_h = \text{span}(v_1, v_2, ..., v_N)$ for the N basis functions v_i.

4. Approximate both forms a and l by new forms a_h and l_h that integrate over the approximate domain Ω_h.

5. Approximate u^g by u^g_h and $\bar{u}$ by $\bar{u}_h \in \mathcal{V}_h$. Express $\bar{u}_h$ as a linear combination of our basis functions:
$$\bar{u}_h(x) = \sum_{a=1}^{N} \bar{u}_{h_a} v_a(x) \tag{1.2}$$

6. Approximate v by v_h and express by a combination of basis functions as above. Simplifying, we find:

$$a_h(\sum_{a=1}^{N} \bar{u}_{h_a} v_a(x) + u^g, v_i) = l_h(v_i), \qquad i = 1, 2, ..., N, \tag{1.3}$$

where this system of equations is either linear or nonlinear depending on the character of the form a_h.

7. Solve the system of equations (1.3) via a linear or nonlinear solver for the unknown coefficients u_{h_a}. These coefficients determine a functional form for our solution function by the formula in (1.2)

This structure applies for the general finite element method applicable to fluid mechanics, magnetism, solid mechanics, and many other partial differential equations. The particular case of computational solid mechanics adds additional complexity due to the highly non-linear character of the form a as well as the need to describe the evolution of history variables such as damage or plastic strain of material points. In addition, fracture problems involve complex topological considerations in an attempt to model the developing fracture surface.

1.2.2 Modern finite element codes

The finite element method is a widely used tool in computational science, and has been researched and developed for the last 40 years. The original finite element codes were largely written in imperative, procedural languages such as Fortran. Although these codes achieved high numerical performance, they were not designed with modularity and extensibility in mind. Even today, some current finite element packages in wide use still have core functionality derived from these original codes, possibly translated from Fortran to C, but with the same design principles. Much progress has been made in improving the parallel performance of finite element codes on large distributed memory clusters. Sandia Labs has developed the SIERRA framework [5], which provides a foundation for several computational mechanics codes, including the explicit code Presto and the quasi-static Adagio. The SIERRA framework provides basic services for parallel multiphysics applications, including mesh refinement, load balancing and communication based on MPI. However, these codes can not yet handle accelerator technologies such as computing on GPUs. In addition, many other new schemes such as discontinuous Galerkin, xFEM, and particle methods such as peridynamics or Smooth Particle Hydrodynamics are only available in very specific new research codes.

The need to integrate all of these features within a single package requires a complete rethinking of the code design and software architecture, as well as the adoption of modern languages and software engineering practices. Object-oriented languages such as C++ provide an acceptable compromise between flexibility and performance as evidenced by the prevalence of C++ in modern codes [1, 6, 7]. Functional, declarative languages (such as OCaml, Haskell, Scheme) are also an appealing environment for writing Finite Element codes. In the first place, mathematical formulations can translate almost directly to code. Secondly, since declarative languages specify what must be done, and not how to do it, they are well suited to parallelization. Unfortunately, the most strict functional languages will not allow side-effects, such as an update of an array in place. An early attempt in the Miranda language showed promise in the clean translation of mathematical formalisms to code, but also revealed the severe performance penalties [8]. However, other functional languages like Haskell have facilities for creating mutable types and may prove more promising.

Most modern C++ finite element codes follow a fairly similar overall structure of classes, which naturally relate to intuitive mathematical objects and operations in the finite element method. Thus, there is a class corresponding to the geometrical mesh for the problem domain, classes corresponding to individual elements, classes corresponding to mathematical solvers, and finally classes corresponding to post-processing and output. The primary differences lie in the boundaries between what the various class definitions encompass and their interactions with each other. We will explore several modern finite element codes by examining their various strategies for determining these boundaries.

One of the first objects involved in a finite element code is an object for meshing and topology. One of the earliest papers on object-oriented design principles for the finite element method [9] provides a MESH class to store all coordinates values and the connectivity information. The class designs in this paper are a fairly direct translation of Fortran design principles, as the MESH class simply wraps the old coordinate array and connectivity table in a C++ class. This object is an active part of calculations, and is passed as a parameter to solvers. Unfortunately, such a design limits a single mesh to be used with a specific type of element, which may not be desirable for multi-physics problems that reuse the same mesh for different equation sets. More recent codes choose to separate the concepts of geometry and degrees of freedom, and define a Mesh as the geometry and topology of the domain as well as a finite element FunctionSpace object for storing degree of freedom and element information (`deal.II` [1], `DOLFIN`,[7]). Other codes provide this separation in principle, if not in name, by including the connectivity map within the Element class (`libMesh` [6]).

The core class of an object-oriented finite element code is the Element class. The exact responsibilities of the Element differ from code to code. In `deal.II` the Element corresponds to a mathematical reference element and provides a polynomial function space along with the associated node functionals on a certain reference domain. The class interface provides methods to determine shape function values and gradients, as well as an enumeration of degrees of freedom per vertex, line, quad and hex (`deal.II` is designed with hypercubes in mind as the reference domain for elements) [1]. A similar approach is found in `libMesh` with their FEBase class for matrix routines and quadrature and derived classes for different function spaces [6]. In both programs, the actual assembly of the system matrix and residual vector is left up to the user. The library code provides iterators for looping over the elements and then shape function values, gradients and jacobians for each element. This allows the library to be used for generic equation sets. However, this approach is not as well suited for solid mechanics since we require a framework for dealing with various material models as well as storage capability for quadrature fields like damage parameters and plastic strains.

One of the advantages of programming a finite element code in C++ is that we can make use of the language's features for inheritance in objects. An abstract Element class can derive into more specific elements and virtual function calls can be used to delegate responsibility to the correct element. Unfortunately, virtual function calls add a layer of indirection that impacts performance. In addition, storing data in local arrays allocated per element can lead to memory fragmentation. One solution is to use templates as a method for delegation instead of inheritance, a solution implemented in `libMesh`. Specific types of finite elements are implemented as template specializations of a templated `FE` class, thus removing the overhead of virtual function calls. A disadvantage of this approach is that different types of elements cannot be put into the same container. Template code can also be more difficult to read and maintain than standard inheritance, and also has an impact on compilation time.

Another approach for improving element performance that has been recently gaining traction is automatic code generation. Given a mathematical expression for the forms in the variational formulation, specific C++ element code for matrix and right hand side assembly can be generated. Using this method allows the application writer to rapidly develop fast finite element solvers for a variety of equations using scripting languages such as Python or specialized languages for specifying mathematics. The DOLFIN code uses this approach and has a large library of elements for use in solving the specified equations [7]. Although this approach allows for the rapid solution of relatively simple PDEs, it is not appropriate for computational solid mechanics since complex material models that include plasticity, viscoelasticity, etc. cannot be expressed in such a simplified form. In addition, more difficult equations may require specific strategies for stabilization, such as the solution of variational forms in a discontinuous function space. The extra stabilization terms that must be added to the form a for different PDEs are the subject of much ongoing research and cannot be easily determined by an automated system.

The original finite element codes would have elements specialized for solving a specific equation, such as linear elasticity. Modern codes are more flexible, but differ on how they handle storing and calling the equations to be solved. Automatic code generators such as DOLFIN or FreeFEM++ [10] have the equations specified by the application writer in a scripting language, and then include them in the assembly routines of their generated element code. The code developed by Besson [9] focuses specifically on solid mechanics, and thus the equation set is the standard field equations for non-linear elasticity, with a constitutive model specified by a material model object attached to an element. More general codes such as deal.II leave the writing of the actual element assembly to the application, and thus have no equation specific code. libMesh takes a compromising approach and provides a hierarchy of System objects that provide the link between elements and equation sets. The assembly routine and associated equation integration is implemented in this System object.

The final step in any finite element method involves the time integration and or solution of a (non-)linear system of equations. In an implicit method, this solution process will require one or more linear solves of a matrix equation. All the previously mentioned codes include some notion of a base Solver class, with inherited solvers for nonlinear problems and time integration. This Solver directs the assembly of any linear system of equations and applies appropriate boundary conditions. Many different software packages are available for the solution of a linear system of equations. Most modern codes include an abstract interface to a LinearSolver class that operates on a SparseMatrix and a Vector for the right hand side. Inherited from this LinearSolver base class are different solvers that can interface to packages such as PETSc [11], MUMPS [12], WSMP [13], etc. Abstracting the solver interface in this manner allows the application writer to choose the most effective solver for the particular problem. In addition, the SparseMatrix class is often an abstract class that has inherited classes for different implementations of sparse matrix storage formats.

1.3 Scope and outline

The remainder of this thesis describes the design and development of a finite element framework flexible enough to implement the technologies discussed in the preceding pages. In Chapter 2, we examine overall code design and discuss the interface and implementation of the various C++ classes involved. Chapter 3 focuses on GPU computing and further explains our motivation for exploring GPU acceleration. There we also present serial and hybrid parallel GPU assembly algorithms and compare computational efficiency to our CPU implementation. Finally, Chapter 4 presents several benchmark problems to showcase the abilities of our code to solve real problems. These include a laser induced shock problem, a wave propagation problem in micro-truss arrays and finally a demonstration of our multiphysics capability with a superelastic material under tensile load.

Chapter 2

Object-Oriented Framework

2.1 Objectives

As stated in the introduction, the original motivation for developing an object-oriented finite element framework is our desire to solve large, complex problems with novel methods and technologies. The limitations of existing codes have prevented us from achieving the full potential of the theoretical schemes available. With this motivation in mind, there are several objectives for the new code that drove our design decisions.

The first objective is to support more general physical problems than finite elasticity. We do not want any implicit assumptions about the nature of our unknown field, such as whether it is a displacement field or its dimension. In addition, we do not want the physical law that we are solving hard coded into our elements. For example, if we simply generalize the unknown field but still retain the traditional elements and material model of computational solid mechanics, we limit ourselves to solving the equation:

$$A(u_i, u_{i,j}, ...)_{ij,j} + b_i = 0 \tag{2.1}$$

where A is a tensor field and a function of the unknown field u (and its derivatives) and b is an applied force. The flexibility gained by generalizing the physics from this equation will allow us to solve more general problems, such as coupled thermal elasticity or superelasticity. At the same time, we do not want the performance penalties associated with an overly general structure. In addition, we want to make developing new applications as simple as possible, and thus our code should offer facilities for quickly setting up and solving common problems, such as finite elasticity.

The second objective is to provide support for a variety of different types of elements, with a focus on space efficiency for hybrid meshes and easy extensibility. By leveraging the object oriented features of C++, we aim to make adding new elements as simple as possible for future development. However, we also want to ensure that we do not sacrifice speed through generality. We envision a simulation capability where the limit in problem size is given by the available hardware and not by the scalability of the software architecture. Therefore, great care must be taken in designing general interfaces to avoid performance penalties for problems scaling to billions of degrees of freedom and hundreds of thousands of processing units.

A third objective is seamless support for emerging parallel hardware, such as massively parallel multi-core CPUs and GPUs. We want to design our element assembly operations in such a way that synchronization and offloading to GPUs is easy to implement in a manner that does not interfere with the structure of the code. Ideally, the application programmer using our framework should not worry about where the code will end up being run, whether on a single computer, a cluster, or a cluster of GPU nodes. In addition, future developers of the framework should not have to maintain two completely separate code branches for GPU functionality and CPU functionality.

Finally, we want to design our solution procedure in a modular way such that it supports a wide variety of solver packages. The ubiquity of linear equations in computational science has led to development of a wide variety of software packages for solving linear systems, such as the aforementioned PETSc [11], MUMPS [12], and WSMP [13]. Each solver has its own strengths and specialties, and an application writer should be able to choose the appropriate solver for their system at compile time based on their needs and the availability of libraries on the current platform. To enable such flexibility, our framework must have a modular interface to a variety of solver packages and matrix storage types.

In the following sections, we will describe how these objectives drove the design of the major modules in our code. Within each section, we will offer some motivation for the scope of the module, and then describe the interface to application writers as well as the internal implementation.

2.2 Topology and meshes

In the background section of the introduction, we described the finite element method in a series of steps. Each code module in our framework is designed to implement one or more of these steps and then interface with the subsequent module. The topology and meshing module deals with step 1 and step 2 of the outlined finite element process:

1. Construct an approximation Ω_h to the original problem domain Ω.
2. Determine a finite element triangulation $\mathcal{T}_h$ for the domain Ω_h, and for each $K \in \mathcal{T}_h$ establish a reference map φ_K.

The topology and meshing module imports a coarse finite element mesh from a file, constructing the appropriate topological information for the given geometrical information and connectivity map. It then provides an interface to access all of this information through iterators over geometric objects, allowing for the construction of the reference maps and element function spaces in the next step.

2.2.1 Interface

In the design of this module, we differentiate between the ideas of geometry and topology within a mesh. The geometry of the mesh deals with the embedding of the mesh in space, and currently only stores the nodal coordinates. The topology of the mesh deals with intrinsic topological structure of the various elements of the mesh, and is completely divorced from coordinates. The traditional connectivity map falls under the category of topology. Our framework uses the concept of a simplicial

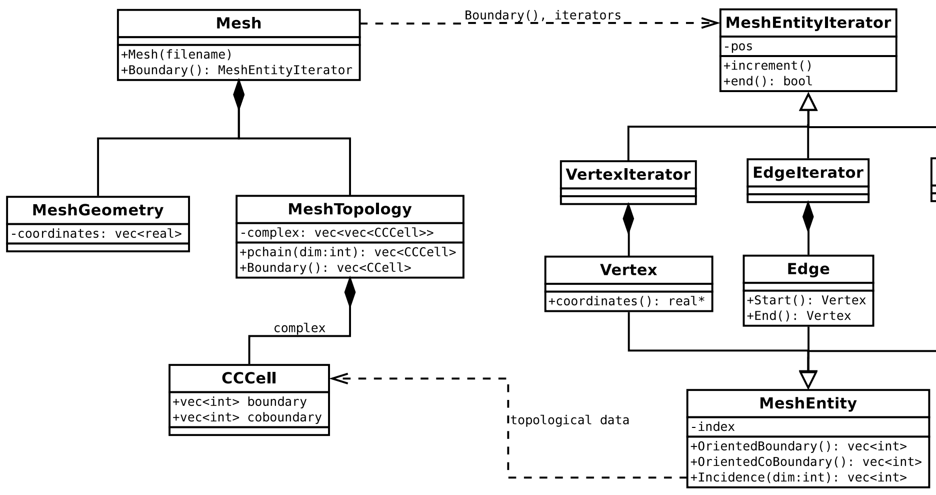


Figure 2-1: The main classes in the mesh and topology module.

complex to store topological information and compute incidence relationships between different topological entities.

An overall view of the main classes in the mesh and topology module is shown in Figure 2-1. The `Mesh` class is essentially a container for a `MeshGeometry` and `MeshTopology`, and provides procedures for loading a mesh from a file. The most important interface to the user is the set of iterators show in the right hand side of the diagram. These iterators are inspired by the work done in Logg, 2009 [14], although our underlying implementation uses a very different representation for the topology. Iterating over the mesh is done through classes derived from a base `MeshIterator`, with specific iterators for vertices, edges, faces, etc. The iterator also contains a corresponding class derived from the base `MeshEntity`, which provides an interface to the topological object pointed to by the iterator. This internal `MeshEntity` is updated as the iterator is moved along the mesh.

Similar to the approach in Logg's paper, our syntax for creating and using a `MeshIterator` is different from the usual STL syntax for iterators. Specifically, iterators are not assigned from a `begin()` function, but rather constructed from either a `Mesh` or a `MeshEntity`. Constructing an iterator on a mesh simply iterates over all the vertices, edges, faces, etc. on the mesh. Constructing an iterator on an entity in a mesh computes the appropriate incidence relationship and iterates over all entities of

Listing 2.1: Demonstrating the use of mesh iterators

```
// adding midpoints to a mesh
for (EdgeIterator edge(mesh); !edge.end(); ++edge) {
  Vertex v1 = edge->Start();
  Vertex v2 = edge->End();

  for (int i = 0; i < dim; ++i)
    coordinates(new_node,i) = 0.5*(v1.coordinates()[i]+
                                   v2.coordinates()[i])

  for (CellIterator cell(*edge); !cell.end(); ++cell) {
    // add new node to connectivity table for given cell
  }
}
```

the specified dimension incident to the given entity. In addition, the end condition is done by directly querying the iterator as opposed to comparison to a special sentinel end iterator. This is done because the end condition can vary greatly depending on the calculated incidence relationship. To illustrate these ideas, an example from the library code is shown in Listing 2.1.

2.2.2 Simplicial complex structure for topology

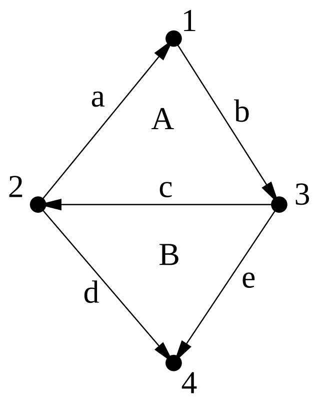
Computing incidence relationships requires topological information about the mesh. For our underlying structure, we use the concept of a cell complex. To elucidate the central ideas, we will briefly summarize simplicial complexes, which are a specific case of cell complexes.

Before we define a simplicial complex, we must define a simplex, or more specifically an n-simplex. An **n-simplex** is an n-dimensional polytope, which is defined as the convex hull of a set of $n+1$ points. For example, a triangle is the convex hull of 3 points and a tetrahedron is the convex hull of 4 points.

Following the definition in [15], a **Simplicial Complex** $\mathcal{K}$ is a set of simplices in $\mathcal{R}^d$ such that:

1. The faces of a simplex $C \in \mathcal{K}$ are elements of $\mathcal{K}$.

2. Given two simplices $C_1, C_2 \in \mathcal{K}$ the intersection of C_1 and C_2 is a either empty



	1	2	3	4
a	+	−	0	0
b	−	0	+	0
c	0	+	−	0
d	0	−	0	+
e	0	0	−	+

	a	b	c	d	e
A	+	+	+	0	0
B	0	0	−	−	+

Figure 2-2: A simple cell complex composed of two adjacent 2-simplices, with vertices (numbers), edges (lower case letters) and faces (upper case letters). A table row indicates a boundary and a column indicates a coboundary. The signs signify incidence with orientation, while a 0 signifies no incidence.

or a common face of both cells, and an element of $\mathcal{K}$.

The **boundary** of a cell $C \in \mathcal{K}$ is the set of faces of C. The **coboundary** of C is the set of cells in $\mathcal{K}$ for which C is an element of their boundary. We can introduce the notion of orientation by assigning a sign to each element in the boundary or coboundary of an element C. These concepts are depicted and further described in Figure 2-2.

While the simplicial complex is restricted only to simplices, the more general cell complex can be formed from any convex polytopes [16]. Thus, our topological data structure can handle hybrid geometries containing quads, hexes, and any other element made from a convex polytope.

2.2.3 Implementation

The current `MeshGeometry` class does little more than store the coordinates of the nodes in a single array. The `MeshTopology` class provides most of the interesting data structures required for the implementation of the iterator interface. The underlying data structure for storing topological information is an implementation of the cell complex described in the previous section.

The actual incidence tables for a typical mesh are very sparse, and we instead store only the oriented boundary and coboundary lists for all the cells. In our implementa-

Table 2.1: Memory usage and topology construction times for our meshing module. The loaded mesh is unstructured and 3D.

Elements	Memory [MB]	Load Time [s]
1920	0.4	0.01
43940	7.9	0.35
160000	28.1	1.41
1097440	189.6	11.20

tion, a cell in the cell complex is a `struct` with a vector of integers for its boundary and a vector of integers for its coboundary. These integers provide a 1-based index for the cells making up the boundary and coboundary, and the sign of the integer indicates orientation. The entire cell complex topology stored in `MeshTopology` is simply a vector of vectors of cells for each dimension. The `MeshTopology` class provides access to the actual vectors of cells for each dimension, but the preferred method to access mesh topology is through the iterator interface.

When a `MeshIterator` is created for a `MeshEntity` structure, we need to compute the incidence relationship for all entities of the dimension of the iterator touching the given mesh entity. The `MeshEntity` class provides a function `Incidence`, shown in Listing 2.2.

Memory usage is a concern when we are storing such extensive topological information. Benchmarks for the memory usage and topology construction time of the cell complex implementation is shown in Table 2.1. Topology information is constructed from the mesh connectivity table, as would be provided by the output of meshing software. The current performance is acceptable, considering that no effort was put in place to optimize the code, and thus there is much room for future improvement in the topology construction time. Most importantly though, the topology import time appears to scale linearly with the number of elements, which gives confidence in the overall algorithm. Extrapolating from this table, we could in theory load a 20 million element mesh on a single 4 GB memory node.

Listing 2.2: Compute an incidence relationship

```
vector<int> Incidence(Cell cell, int incidence_dim)
{
  // temporary storage for incidence at every dimension
  vector<set<int>> incidence;

  // lower dimensional incidence
  if (cell.dim > incidence_dim) {
    for (face in cell.boundary)
      incidence[cell.dim-1].insert(face.id());
    for (int i = cell.dim-1; i > incidence_dim; --i)
      for (id in incidence[i])
        for (face in Cell(id).boundary)
          incidence[i-1].insert(face.id());
  }
  // higher dimensional incidence
  else if (cell.dim < incidence_dim) {
    for (face in cell.coboundary)
      incidence[cell.dim+1].insert(face.id());
    for (int i = cell.dim+1; i < incidence_dim; ++i)
      for (id in incidence[i])
        for (face in Cell(id).coboundary)
          incidence[i+1].insert(face.id());
  }
  // equal dimensional incidence
  else {
    for (face in cell.boundary)
      for (coface in face.coboundary)
        incidence[cell.dim].insert(coface.id());
  }
  return vector(incidence[incidence_dim]);
}
```

2.3 Finite elements module

The next module in our new framework involves the actual finite element method and collections of elements. Following our mathematical outline, the steps handled by the finite element module are:

3. Choose a function space on each $K \in \mathcal{T}_h$, thus establishing an approximate function space $\mathcal{V}_h$ to the original function space $\mathcal{V}(\Omega_h)$. Given N degrees of freedom, we have that $\mathcal{V}_h = \text{span}(v_1, v_2, ..., v_N)$ for the N basis functions v_i.

4. Approximate both forms a and l by new forms a_h and l_h that integrate over the approximate domain Ω_h.

Before describing the interface and implementation of the elements module, we will consider the performance implications of various element containers as that will drive our later design decisions.

2.3.1 Fine grained vs. coarse grained containers

The advantages of using an object-oriented language for developing our finite element code would be lost if we did not make full use of features like inheritance via class derivation and polymorphism via virtual methods. The elements themselves make perfect candidates for inheritance due to the hierarchical nature in their structure. Basic 1st and 2nd order tetrahedra can derive from a continuous Galerkin abstract element that implements some of the shared functionality such as generalized residual assembly. In addition, all elements can inherit from a base abstract Element class that determines the interface for handling allocation and assembly. Then, we can put all of our elements into a single container and take advantage of virtual function calls to save us from cumbersome switch statements. Such behavior is present in many older C codes as well, implemented through structs and function pointers, whereas in C++ the language provides built in support for these constructs.

Unfortunately, calling virtual functions has an overhead. Two extra memory fetches are required, one to get the class virtual table (vtable) address and one to get the call address of the virtual function. For a non-virtual function, the call address can be resolved at compile time. As an aside, we note that this penalty is not additive for extra layers of inheritance. For example, if we have a class C deriving from a class B that derives from a base class A, a virtual method defined in all the classes will incur the same penalty when called on a C object as when called on a B object. Usually, the overhead of a virtual method call is negligible - unless there are millions of virtual method calls in a loop like in a finite element code. An additional, and perhaps greater, concern is heap fragmentation. For example, if we have an element object for each element in the mesh, and each of these classes allocates its own arrays for storing shape function values, stresses, etc., then these arrays may become scattered through the memory heap. Running an assembly sequentially across the elements will then incur a great penalty due to poor cache behavior from random access into the heap.

To quantify the impact of this overhead, we ran a performance test on a toy application designed to model possible finite element container implementations in C++. This application provides a simple class hierarchy (Figure 2-3) to model the possible inheritance structure of elements in a finite element program. The abstract class `Element` defines an interface to the virtual function `Expensive` that performs a simple mathematical operation (different for each element) on the array of data in the element, producing a single double as a result. We are interested in running `Expensive` on a large set of data, with an array of different `Elements` to perform the operation. Three possible implementations have been produced. The fine grained implementation has a single `Element` class for every calculation element and also allocates the data array within the element. The pool implementation is similar to fine in that there is an `Element` object for each calculation element, except that the data array is allocated for all elements ahead of time and individual elements index into this coherent array. Finally, the coarse grained implementation uses aggregating element sets, where a single class presents an interface to many elements of the same type. The data for a single element type is stored as a pool within this set.

We ran the test program for 1,000,000 elements evenly distributed among the four

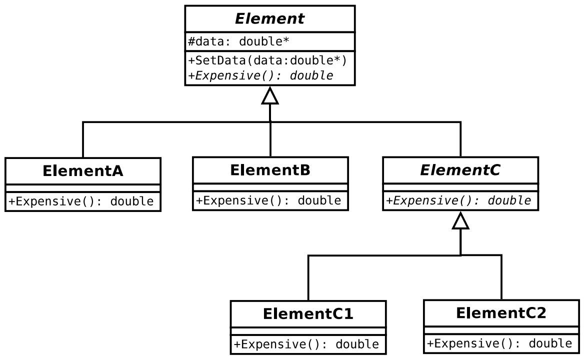
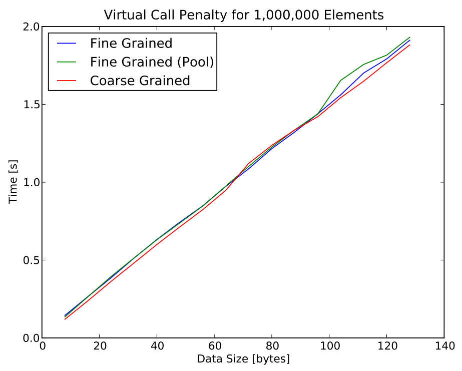


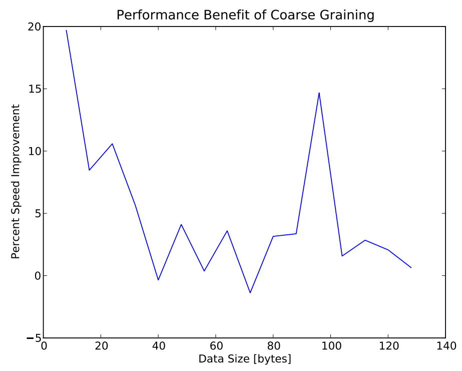
Figure 2-3: Class hierarchy for virtual overhead test application. The virtual function `Expensive()` performs an expensive calculation on an array of data.

different types and varying data sizes per element. The test was run on a 64-bit machine, with the code compiled by `gcc` with `-O3` optimization enabled. Each test was run 100 times, with the fastest and slowest times thrown out. The timing results are presented in Figure 2-4a. Surprisingly, both fine grained implementations have the same performance. The fragmentation penalty we expected does not exist in this simple test. However, caching behavior is extremely complex and difficult to predict, and this demo program is not necessarily representative of real world behavior. The coarse grained implementation has the best performance for all data sizes. The penalty for the virtual function call is approximately 20% for the smallest data sizes (corresponding to relatively little computation) and decreases to be negligible for the largest data sizes (Figure 2-4b). As expected, more intense computation amortizes the extra overhead of a virtual function call.

These results are encouraging, but in themselves are not a convincing argument for using coarse grained element sets. However, using element sets affords us flexibility in the implementation of functions such as residual assembly. Specifically, it aids us in developing multi-threaded and GPGPU codes by capturing details of synchronization in the element set itself. Thus, we can use the same exact interface regardless of whether our code is running sequentially, with shared memory multi-threading, or on a GPU. Using fine grained classes for elements would require a separate interface for GPU codes as well as more invasive synchronization capability in the assembly routines. This approach allows our assembly routine to be blind to the details of how



(a) Performance Results



(b) Coarse Grained Improvement

Figure 2-4: Performance results from a test on a simple class hierarchy to determine the penalty of virtual function calls and memory fragmentation.

our element calculations are performed.

2.3.2 Interface

With the advantages of using element sets established, we can now describe the user interface to the finite elements module. This module comprises several related classes that provide the functionality of a spatial finite element discretization. Specifically, the finite elements module allows us to establish a function space over our domain and then compute residual vectors and stiffness matrices for the weak form of our problem given an approximation in that function space. The primary classes involved are the abstract base `ElementSet` and its derived classes, the `Mapping`, a `DoFMap`, the `FunctionSpace` and the `WeakForm`. An overall picture of the relationships between these classes is shown in 2-5.

The primary interface to the element module is the `FunctionSpace` class. This class captures the idea of an assembly of individual finite elements over a spatial domain, essentially the second part of step 3 in our original mathematical description. Thus, the `FunctionSpace` provides us with support for function interpolations over the domain, and thus is a basis for the `NodalField` and `QuadratureField` classes. Its interface provides functions for interpolating from a `NodalField` to a `QuadratureField` or extrapolating a `QuadratureField` to a `NodalField`. The latter

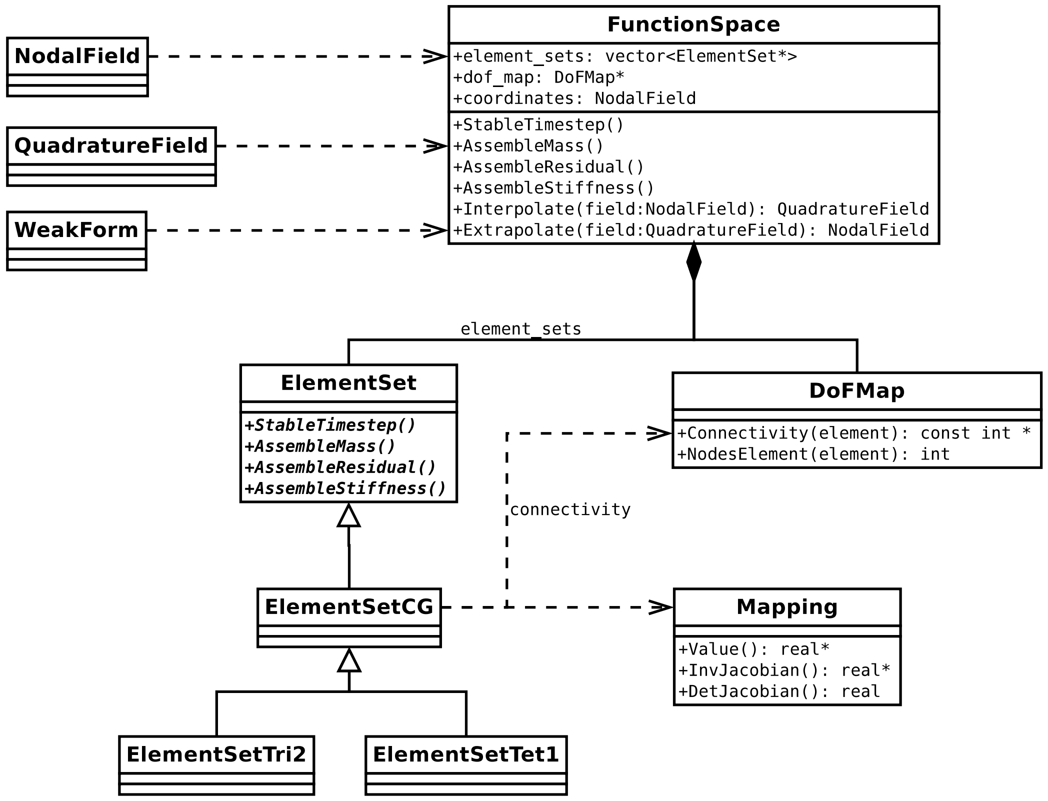


Figure 2-5: Primary classes and relationship in the elements module.

operation is not well defined since we may be extrapolating from a discontinuous to a continuous function space. However, an approximation suffices since this function is only used for data output. Finally, the `FunctionSpace` has an interface to our basic finite element assembly functions, namely those for assembling the residual vector, the mass matrices, and a stable timestep.

Importantly, the `FunctionSpace` deals only with spatial interpolation and has no knowledge of the equations being solved. Information about the equation is captured in the abstract `WeakForm` class. The purpose of this class is to provide stiffness matrix and residual calculations for the weak form of our equation of interest at points in our domain. Derived classes from this base class provide the actual functionality; for example, most of our calculations use the `MechanicsWeakForm`. Although the weak is used heavily by the elements module, it is actually owned by a system, which shall be discussed in the next section.

The remaining classes in the element module, including the element sets, should not be dealt with directly in application code. All issues of allocation or calling assembly operations are done through higher level classes such as the `FunctionSpace`. An application writer only needs to worry about specifying which element or elements to use for a given mesh.

2.3.3 Implementation

While the `FunctionSpace` class provides the main user interface to the spacial discretization, all the actual calculations are performed by the element sets. An element set is a container for a set of elements, all of the same type and discretizing the same equation. A simple interface is provided in the abstract `ElementSet` class, which simply requires that the derived element sets be able to perform assembly operations for our residual vector and mass and stiffness matrices. All other details of initialization and interaction are left to the discretion of the derived class.

Many of our elements are derived from the `ElementSetCG` class, which inherits from `ElementSet`. This class is for continuous Galerkin (CG) elements, and provides many of the support features in common for CG elements such as residual and stiffness vector assembly. The CG element base class also stores the shape function and jacobian values at quadrature points for later use in integrating functions. These values depend on the actual element being used, and their computation is done in the constructors for derived classes, such as `ElementSetTri2`. To compute shape function values, these derived classes need to know how the reference element for the element set is mapped into the particular element under consideration. This knowledge is provided by the `Mapping` class, which captures the idea of the element reference map φ_K referenced in our mathematical description.

Tying together element function spaces requires that we have a connectivity table that relates element local degrees of freedom to global degrees of freedom. This functionality is provided by the `DoFMap` object in our function space. This object is responsible for holding the connectivities of all the elements in our element sets, which may have different numbers of nodes. In addition, this class can provide an interface for degree of freedom renumbering algorithms to reduce the bandwidth of the global stiffness matrix.

2.4 System of equations and fields

One of the objectives of our code is support for multi-physics problems involving multiple unknown fields. For example, we may want to do a coupled simulation of thermal elasticity with a displacement field for the mechanics as well as a temperature field. Another example is the simulation of a shape memory alloys, where we may want to include the volume fraction of martensite as an additional field. To support different physical problems, we introduce the `System` class that acts as a container for the unknown fields in the problem as well as the weak form of the system equations. The `System` then interfaces directly into solver and integrator classes to determine the actual solution. A single simulation can also use multiple systems. For example, a shape memory alloy calculation can proceed in an staggered fashion, and use a system for the mechanics problem and another system for the volume fraction problem. These multiple systems can share the same `FunctionSpace` object, reducing the memory requirements.

2.4.1 Interface

To interface well with various solvers (described in the next section), system classes are derived in an specific inheritance tree that describes their mathematical structure (Figure 2-6). All classes derive from a common base, the `System` class, which provides a single interface for output classes. An application programmer constructs a system from a previously initialized `FunctionSpace` object, allocates the fields, and then continues to create the solver object. The system by itself cannot solve any equations, it is essentially a wrapper for the mathematical fields involved in the problem as well as the actual equations being solved.

Another important aspect of the interface is the accessors to unknown fields. These are specified in the abstract `NonlinearSystem` and `HyperbolicSystem` classes. For example, the `NonlinearSystem` class has the function `u()` that provides access to the

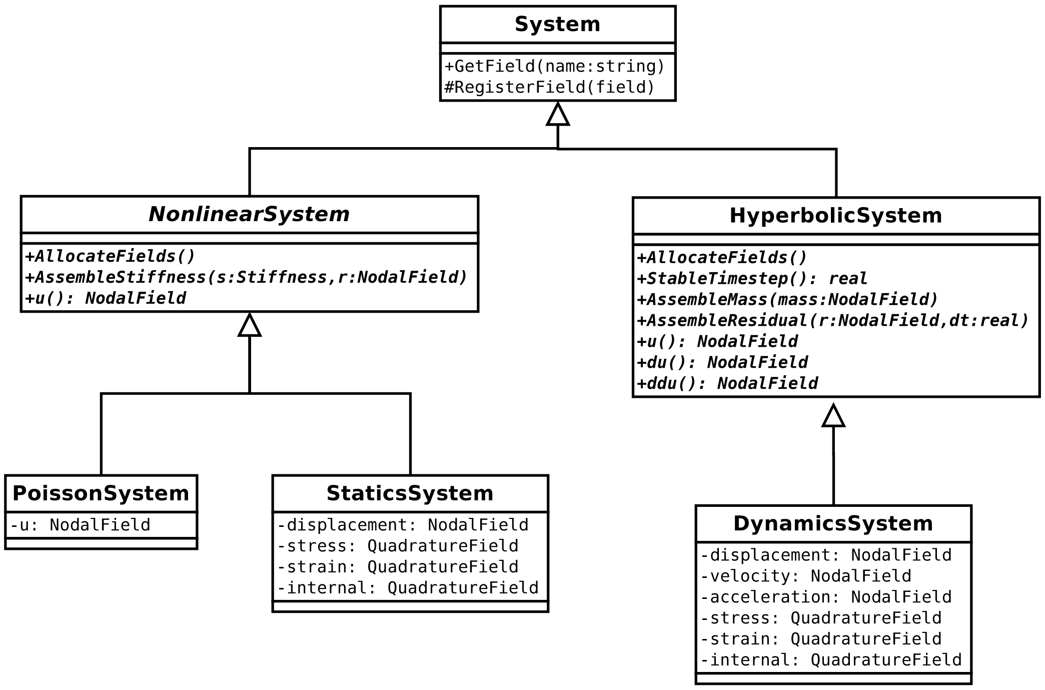


Figure 2-6: Interface to systems module

unknown field, whatever it may actually be called within the class itself. Any class deriving from a `NonlinearSystem` must implement this function and have it return the primary unknown. A `NonlinearSolver` will then use this interface when solving the system, as detailed in the next section.

Output is performed via the `SystemWriter` class. This class allows an output writer to be created for a given system, and allows specified fields to be dumped after performing a solver step. An output writer is initialized with the name of the output file, the system to write, and the type of writer to use (VTK, Tecplot, etc.). An example from of a user application of creating a system and an output writer is shown in Listing 2.3.

2.4.2 Implementation

The `System` class contains two functions for internal use, `GetField` and `RegisterField`. The `GetField` function takes a string for the name of the field and returns a const reference to the `NodalField` in the system, if it exists. This functionality is implemented through an STL map between strings and `NodalField` references. The

Listing 2.3: Initializing a system for a statics problem

```
#include <systems/statics_system.h>

int main() {
  // function space created here

  // setup system
  summit::StaticsSystem tension_problem(function_space);
  tension_problem.AllocateFields();

  // create and initialize a solver for the system
  // ...

  // create a VTK output writer
  summit::SystemWriter writer(``output'',
                              tension_problem,
                              function_space,
                              summit::MESH_WRITER_VTK);
  writer.AddOutputField(``displacement'', ``stress'');

  // solve the system for a load increment
  // ...

  // output the solution for load increment 0
  writer.Write(0);
}
```

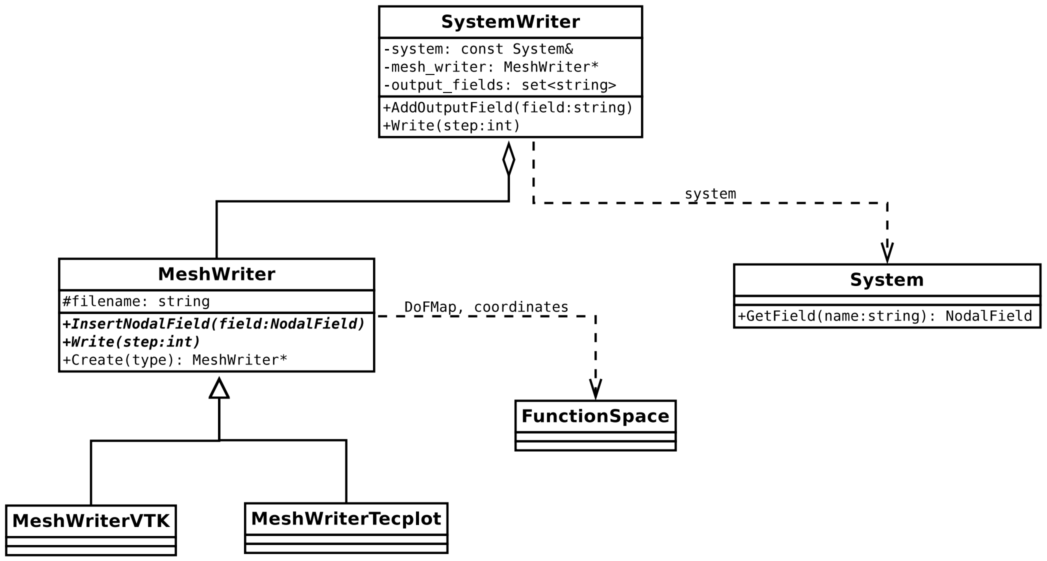

SystemWriter
-system: const System&
-mesh_writer: MeshWriter*
-output_fields: set<string>
+AddOutputField(field:string)
+Write(step:int)
system
MeshWriter
#filename: string
+InsertNodalField(field:NodalField)
+Write(step:int)
+Create(type): MeshWriter*
DoFMap, coordinates
System
+GetField(name:string): NodalField
FunctionSpace
MeshWriterVTK
MeshWriterTecplot

Figure 2-7: Relevant classes for system output

`RegisterField` function registers a given `NodalField` or `QuadratureField` with the `System` so that it can add it to the map. The constructor for any of the derived systems must call this function on all of its fields so that they can be referenced and outputted.

All output is done through a `SystemWriter` class, as shown in the example code of the previous section. The `SystemWriter` is associated with a specific `System` instance, and holds a const reference to it as one of its members. The writer records which fields should be outputted upon a call to the `Write` function in an STL set of strings. When output is requested, it iterates through this set, gets the field from the system and outputs it using a mesh writer. A mesh writer is a class that can output a given `NodalField` on a `FunctionSpace`. Different mesh writers derive from a base class, allowing for implementation of output in various formats such as VTK, TecPlot, NetCDF, etc. The required mesh writer is created upon instantiation of the `SystemWriter` via a static factory method, based on whatever type of output is required by the user. A summary of these concepts is shown in Figure 2-7.

The base `System` class provides no further functionality than the basic support for output and field access. Derived classes for specific equation sets require extra code for proper initialization of fields and assembly of vectors. For example, the

`StaticsSystem` class sets the initial gradient of displacement to the identity tensor, and resets to zero stress. Although `StaticsSystem` has a function for assembling the stiffness matrix, the actual process of assembly is delegated to the element sets through the associated `FunctionSpace`, as described in the previous section. The primary purpose of the system is to simply be a container for the unknown and derived fields in the problem.

2.5 Solvers and integrators

Once we have a spatial discretization for our system of equations, we will need to numerically integrate or solve our system. This is the final step of our mathematical formulation:

7. Solve the system of equations (1.3) via a linear or nonlinear solver for the unknown coefficients u_{h_a}. These coefficients determine a functional form for our solution function by the formula in (1.2)

For implicit solves, the stiffness matrix is very sparse, and we use a special sparse matrix to store it. Some of the early finite element codes were directly tied to an internal sparse direct solver. While this close coupling makes development easy and the code efficient, it limits the code to the capabilities of the included solver. An objective of our new framework is a general interface to linear solvers and integrators so that a user can choose the best solver for their problem and the code can keep up to date with new technologies while avoiding performance penalties. As better solvers are developed, we can simply add wrapper interfaces around them and integrate them into our code.

2.5.1 Interface

An application writer must choose the appropriate solver to create for their system. For example, in solving a `StaticsSystem` an application writer could use either a

Listing 2.4: Creating a solver for a statics problem

```
// function space created previously
summit::StaticsSystem tension_problem(function_space);
tension_problem.AllocateFields();

// create and initialize a solver for the system
summit::LinearSolver solver(&tension_problem);
solver.Init();

// create boundary conditions
// ...
// setup nodal fields boundary and forces here
// ...
solver.SetBoundaryConditions(boundary, forces);
solver.AllocStiffness(function_space);

// solve the static loading step
solver.Solve();
```

LinearSolver or a NewtonSolver depending on whether or not small or large displacements are being considered and if the material model is linear or nonlinear. Using a completely inappropriate solver, such as a ExplicitNewmarkIntegrator for a StaticsSystem will create a compile time error, since the integrator expects a HyperbolicSystem (see Figure 2-6 for the hierarchy of systems). Boundary conditions types and forcing values are set through the solver interface, as they are necessary for the solution of a static or time step. We can solve the simple statics problem from Listing 2.3 with a solver as shown in Listing 2.4

The allocation of the stiffness matrix in an implicit solver must be done after boundary conditions are specified, since we use static condensation in our stiffness matrix for Dirichlet degrees of freedom. The boundary conditions can be specified either in application code, or loaded through an external file.

2.5.2 Serial and parallel solver implementation

The solver implementation consists of several layers. At the top layer are classes derived from the base Solver class that are designed to either integrate or solve a given System. Aside from the explicit integrators, all solvers eventually require some sort of linear solve. Thus, at the bottom layer is the SparseMatrix class that provides the

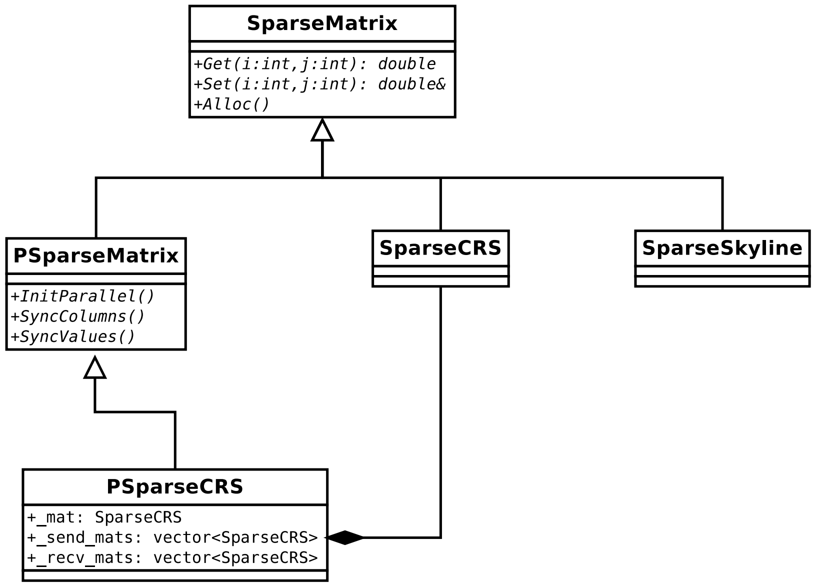


Figure 2-8: Class heirarchy for SparseMatrix

structure for a large serial or parallel sparse matrix. Finally, in between we have the `Stiffness` class that adds in additional knowledge about the finite element method on top of a `SparseMatrix`. This additional knowledge includes functions to initialize the non zero structure from a given mesh connectivity as well as the equation map structure for handling the static condensation of the Dirichlet boundary conditions.

In a serial run of our code, the functioning of these classes is relatively straightforward. The user application requests a solve or a time step integration from a `Solver`. The solver then goes through the appropriate algorithm, whether it is Newmark integration or Newton-Raphson. It uses the interfaces described in the systems section to access the unknown field. For example, the Newmark solver gets the displacement, velocity and acceleration from the system. It then performs the predictor step, calls the residual assembly function on the system and finally performs the corrector step. The system class knows how to perform the assembly operation, but the responsibility for correctly modifying the physical fields are the responsibility of the solver.

Parallel Solver

The implementation of a parallel solver class presents many additional difficulties. There are several readily available parallel direct and iterative solvers that we can

Listing 2.5: Negotiation of stiffness matrix row ownership

```
int node_start, node_end;
// first node starts the process
if (myPID == 0) {
  node_start = 0
  node_end = max(global_ids);
  MPI_Send(&node_end, myPID+1);
}
// each following node proceeds in turn
else {
  MPI_Recv(&node_start);
  node_start++;
  node_end = max(global_ids, node_start);

  if (myPID < nProcessors-1)
    MPI_Send(&node_end, myPID+1);
}
```

interface with. However, we require a parallel sparse matrix object that can handle assembly and proper synchronization between processors. The class heirarchy for deriving the parallel sparse matrix is shown in Figure 2-8. The `PSparseMatrix` class inherits from a generic `SparseMatrix`, which allows us to seperate the parallel implementation from the user interface.

The `PSparseCRS` class implements a specific type of parallel sparse matrix based on the Compressed Sparse Row serial matrix. Each processor in the cluster takes ownership of a set of rows in the matrix. This division corresponds to the input required by most parallel solvers. The matrix values for these rows are stored as a `SparseCRS` member in the `PSparseCRS` class, and the row offset is stored as an integer. In addition, the `PSparseCRS` object has a vector of send matrices and a vector of receive matrices to buffer values that must be later synchronized with other processors.

The row partitioning is negotiated during initialization of the stiffness matrix. The nodes in the mesh have already been labeled with global IDs from the parallel mechanics module. The row partition negotiation algorithm is shown in Listing 2.5. It proceeds serially through the processors, and has run time $O(N)$ where N is the total number of nodes in the mesh.

In addition, we must initialize an equation map for the stiffness matrix, which is done simultaneously with node ownership assignment. This equation map provides a mapping from degrees of freedom to equations in the stiffness matrix, allowing for static condensation of Dirichlet boundary conditions. Note that the concept of an equation map is tied to the stiffness matrix, and the parallel sparse matrix underneath deals only with rows and columns.

After row numbers are assigned, the sparse matrix initializes its send and receive matrices. Each processor has a communication map that specifies which processors it communicates with based on the manner of the domain decomposition. Each processor informs its neighbors of its row ownership via non blocking sends. Simultaneously, it receives the row ownerships of all of its neighbors. These values determine the range of validity for its send matrices.

The parallel sparse matrix is initialized from the `DoFMap` in a similar manner to the serial sparse matrix. However, in the parallel case we have the added complexity of initializing the send and receive matrices. In the serial case, the `SparseCRS` matrix is passed a vector of non zeroes per row for allocation. In the parallel case, we must pass the number of non zeroes per row in the ownership range of the current node, as well as the number of non zeroes per row in the range of each neighboring node. Once the send matrices on a processor are initialized, it passes the non zero information for each send matrix to the appropriate neighbor so that the neighbor can initialize its receive matrix. Since each send matrix corresponds to a neighbor, the communication time here is only dependent on the number of neighbors for a single processor. Thus, total allocation time should be linear in the size of the matrix, just as for the serial case.

Setting values in a parallel sparse matrix is done through the same `Get`/`Set` interface present in the serial sparse matrix. Although the interface is the same, there are limitations to the parallel implementations of these functions. The `Get` function can only access values stored locally on the calling processor, i.e. in the row ownership range of that processor. The `Set` function can only write values that are either stored locally on that processor, or stored on a neighboring processor. Note that this is simply a limitation of this particular implementation, and not the interface itself. These limitations are acceptable, since for our finite element implementation these are all the accesses we need.

To optimize the amount of communication, synchronization occurs only when specifically initiated by the user. Thus, a `Set` command called with an index not owned by the processor will be written to one of the send matrices. In the synchronization step, all send and receive matrices are exchanged via non-blocking MPI calls. Since we have already established the matrix structure in the initialization, we only need to exchange the actual matrix values. After all the matrices are exchanged, a processor iterates through its receive matrices and adds the components to its own matrix. Since the maximum number of neighbors a processor has is ultimately a geometric property that is a function of the dimension of the space, our synchronization can be completed in $O(N/P)$ time, where N is the number of nodes in the mesh and P is the number of processors.

Complicated material responses can produce very ill conditioned matrices that prove difficult for iterative solvers. Thus, we are interested in recent advances in parallel direct solvers, such as the WSMP solver [13]. To this end, we test our solver framework on a simple implicit problem, namely the crushing and buckling of an aluminum sandwich panel (Figure 2-9). Our test mesh has 104 thousand elements and 0.5 million degrees of freedom. This mesh produces a matrix with 17.5 million non zeroes. We measure the matrix assembly and solve times for 64 solver threads, using from 1 to 4 solver threads per processor. The results shown in Figure 2-10 indicate that the parallel assembly algorithm detailed in the previous paragraphs has perfect linear strong scaling. The solver does not scale as well as the assembly, but the solver is provided by an external module that can be replaced and updated as technology improves.

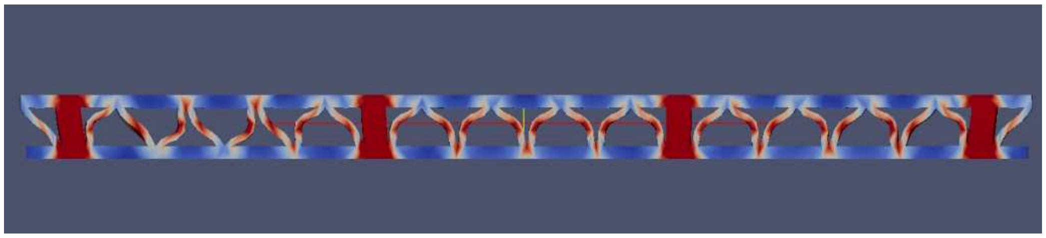


Figure 2-9: Buckling of a webbed aluminum panel under a compressive load. Colors indicate stresses in the panel, with red indicating compressive stress and blue tensile stress.

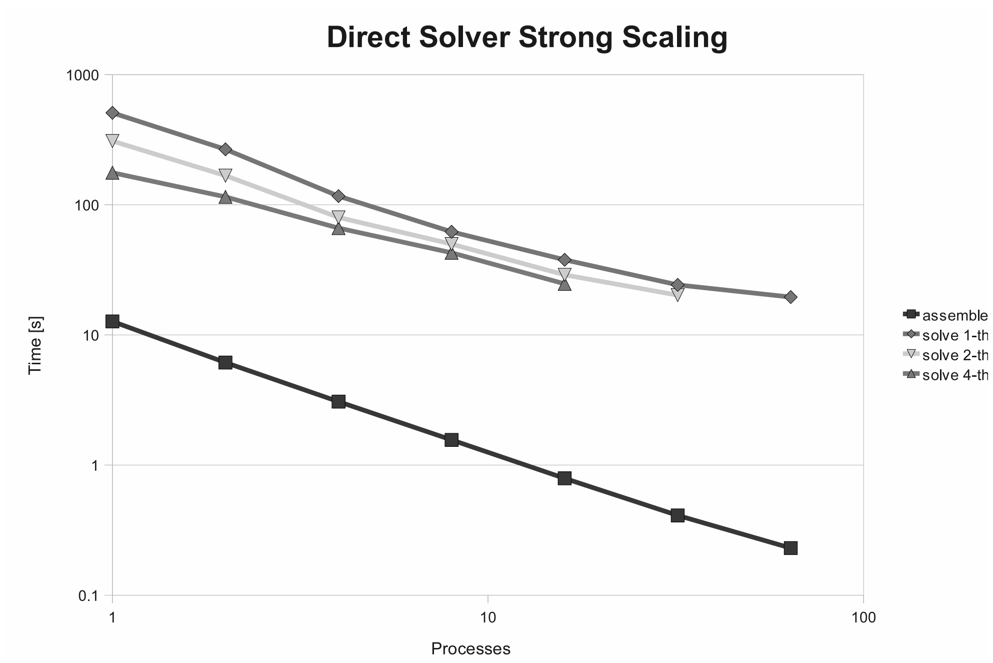


Figure 2-10: Strong scaling for direct implicit solver, showing assembly time and solve time for varying numbers of threads per process.

Chapter 3

GPGPU Acceleration

3.1 Motivation

General Purpose processing on Graphics Processing Units (GPGPU) has made enormous strides in recent years. Many applications have been written to perform faster computations at lower cost for problems in the financial industry, geophysics, and medical imaging. Problems that would take hours on a distributed memory cluster can be solved within minutes on a desktop with a sufficiently powerful GPU. The HPC community has also become very interested in GPGPU. In November 2010, three of the top ten computers on the Top500 list included GPU hardware accelerators, including the fastest computer - Tianhe-1A [17].

There are several reasons for the growing popularity of GPGPU acceleration. In the first place, GPUs are widely available and relatively cheap in terms of GFLOPs performance per dollar, largely due to the demands of the gaming industry. Most modern desktop computers and a growing number of laptop computers have dedicated video accelerators capable of general purpose calculations. In addition to these graphics accelerators, dedicated compute processors based on GPU chip design have become available in the last few years.

Although GPU hardware has been available since the mid to late 1990s, the software and hardware to perform GPGPU has only recently become available. The original GPUs focused on accelerating 3D rasterization, and the hardware was fully customized for rendering triangles and mapping textures. The introduction of fully programmable pixel and vertex shaders first enabled general purpose computation. Between 2003 and 2005 the GPGPU movement began gaining momentum, taking advantage of programmable shaders to perform scientific computations with graphics hardware. However, shader languages were specialized to graphics operations and programming scientific applications required a lot of specialized hardware knowledge. The introduction of frameworks specific for GPGPU, such as NVIDIA's CUDA ([18]) in November 2006, greatly lowered the knowledge barrier . The CUDA framework provides a set of compilers and libraries that allows general C programs to be compiled for execution on the GPU. Since its introduction, there has been an explosion of GPU applications, particularly in molecular dynamics, geophysics, computational fluid dynamics and medical imaging [19], [20].

One major reason for the heavy interest in GPUs is the suitability of the hardware for scientific computing. In the last few years, CPUs have hit a frequency performance barrier, where increasing the clock frequency to improve performance has a prohibitive cost in power consumption. This barrier has driven the development of multicore design, with most laptops and desktops in 2010 containing multicore chips. However, a CPU core is a heavyweight core that can execute an operating system and is optimized for sequential performance. By contrast, a GPU is composed of hundreds of lightweight cores optimized for arithmetic performance with high memory bandwidth [21]. In NVIDIA GPUs, these lightweight cores are known as stream processors and they together execute a single instruction for a series of threads in a model known as SIMT (Single Instruction, Multiple Thread). Each thread operates on a distinct piece of data, and thus the model is well suited to highly data parallel applications.

Finally, the theoretical arithmetic performance of GPUs has exceeded the performance of CPUs in the last several years. For example, a single NVIDIA 8800 GTX from 2008 achieves 330 billion floating point operations per second (GFLOPs) and a peak memory bandwidth of over 80 GB/s. This performance is significantly more powerful than even high end CPUs [22]. Modern GPUs in 2011 can achieve upwards of 1 TFLOPs performance on a single chip. The almost order of magnitude gap between GPU and CPU cores in pure arithmetic performance has generated great interest for computationally dominated applications.

Although CUDA was one of the first GPGPU architectures to be introduced, there are several others currently available. AMD has gone through a series of architectures, including the most recent FireStream SDK, formerly known as Close To Metal. In addition, there is the open industry standard of OpenCL (Open Compute Language). We have chosen to work with CUDA for our application due to its widespread adoption and the current performance edge CUDA enjoys over OpenCL on NVIDIA hardware.

GPU accelerated finite element applications have been around for almost as long as GPGPU. In 2005, Goddeke et al. published one of the original papers on accelerating a simple Poisson problem with a GPU, although their primary performance gains were in the acceleration of the iterative linear solver [23]. In the geophysics community, Komatitsch et. al. implemented a higher order elastodynamics solver for linear anisotropic materials [24]. Linear elasticity is also treated by Goddeke in 2009 [25], along with strategies for encapsulating the acceleration code in the library with minimal impact on the user. Nonlinear finite elasticity using a neohookean material was first treated by Taylor in 2008 with the idea of applying real-time computations of tissue mechanics for surgery planning [26].

3.2 Serial FEM with CUDA

GPU compilers have not yet advanced to the state of CPU compilers in terms of performance optimization. Thus, achieving high performance in GPU code requires an understanding of GPU architecture and ways to best exploit its features. We will briefly describe the salient features of GPU hardware to enable our later discussion on performance optimization. A modern NVIDIA GPU is composed of hundreds of small Scalar Processor (SP) cores capable of executing arithmetic instructions. These SP cores are organized into groups known as Streaming Multiprocessors (SMs). A SM contains eight SP cores, an instruction unit as well as a local shared memory cache. The SIMT model is evident in this arrangement - the instruction unit of an SM issues a new instruction and the eight SP cores all execute it for their separate threads, presumably operating on separate data. As long as there are no divergent conditional statements in the program execution and none of the threads stall on reading memory, the program execution will fully utilize the compute cores and achieve near the theoretical peak arithmetic performance.

The CUDA programming model is driven by the GPU architecture. A single program thread runs on a SP core. On the software side, these threads are grouped together into blocks. A block runs on a single SM, allowing for block-level synchronization commands such as `__syncthreads()`. The SM handles all the overhead for scheduling the block threads on the SP cores and implements very fast hardware thread context switching. An SM can execute one, or as many as eight blocks depending on the configuration of the execution grid and the register and shared memory usage of the thread blocks. Both registers and shared memory use the local shared memory cache located on the SM. A set of blocks together forms a grid, which is the unit of parallel execution for a program kernel in CUDA. When executing a parallel kernel, the programmer specifies the dimensions of a block and the dimensions of the grid. Further information on these concepts can be found in the CUDA Programmer's Guide [18].

With this knowledge in mind, we shall examine how to implement an explicit finite element method for solid dynamics. The field equation for non-linear solid dynamics is:

$$P_{iI,I} + \rho_0 B_i = \rho_0 a_i \tag{3.1}$$

where P_{iI} is the first Piola-Kirchhoff stress tensor, B_i is any body force, and a_i is the acceleration field. To integrate this equation, we use an explicit, central-difference time integration scheme. The algorithm for computing a new time step for the spatially discretized problem is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \triangle t\,\mathbf{v}_n + \frac{\triangle t^2}{2}\mathbf{a}_n \tag{3.2}$$

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \frac{\triangle t^2}{2}(\mathbf{a}_n + \mathbf{a}_{n+1}) \tag{3.3}$$

$$\mathbf{M}\mathbf{a}_{n+1} + \mathbf{f}^{\text{int}}_{n+1} = \mathbf{f}^{\text{ext}}_{n+1} \tag{3.4}$$

where the subscript n refers to time t_n; $\mathbf{x}$, $\mathbf{v}$ and $\mathbf{a}$ refer to the nodal discrete deformations, velocities and accelerations; $\mathbf{f}^{\text{int}}$ refers to nodal internal forces from the divergence of the Piola-Kirchhoff stress; $\mathbf{f}^{\text{ext}}$ refers to any nodal external forces and $\mathbf{M}$ refers to the mass matrix [3]. Via mass lumping of the elemental mass matrices our mass matrix is diagonal, and thus no equation solving is necessary for the explicit step.

This time discretization translates into 3 basic steps for our algorithm. In the predictor step, nodal deformations and velocities are updated based on (3.2) and (3.3). Next, we calculate a residual force as a function of the updated deformations, defined as $\mathbf{r}_{n+1} = \mathbf{f}^{\text{ext}}_{n+1} - \mathbf{f}^{\text{int}}_{n+1}$. This residual calculation requires calling the material constitutive law with the updated deformation gradient based on the predicted deformations. The stress from the constitutive evaluation is then used to integrate the virtual work in the domain, which allows us to determine the nodal internal force vector. Combined with the external forcing vector, this allows us to calculate the new residual vector. Finally, the corrector step calculates new accelerations from (3.4) and corrects velocities from (3.3).

Typically, our dynamic simulations are run for from thousands to hundreds of thousands of explicit timesteps. Within a single explicit step, most of the time is spent in assembling the residual vector. For example, when running a 160,000 element mesh on a single core, the CPU version of our code spends, normalized per element, 0.017 μs on the predictor step, 2.42 μs on the residual calculation and 0.034 μs on the corrector step. This example simulation uses a neohookean material model, more complicated materials models such as J2 plasticity will require even greater time on the residual calculation with no increase in predictor or corrector time. Thus, following Amdahl's Law, our acceleration efforts should focus on increasing the speed of the residual vector calculation.

Fortunately, the residual calculation is very data parallel and thus well amenable to acceleration on GPUs [27]. However, before attempting to parallelize our explicit update algorithm via the shared memory paradigm for GPUs, it is necessary to understand the data dependencies involved. Figure 3-1 shows the flow of data during residual calculation. Since we use unstructured meshes, the element local vectors access values from scattered locations in the nodal displacement vector. They also write to scattered, and possibly overlapping locations in the nodal residual vector. These locations are determined by the connectivity table of the mesh. There are two consequences to this data flow. In the first place, we will have an unavoidable penalty in reading nodal data from scattered memory locations due to the poor caching behavior of random access. This penalty can be avoided by either using structured meshes, which is undesirable for problems with complex geometry, or by using a different numerical scheme such as discontinuous Galerkin methods [28]. A second consequence is that we will need some method of synchronization for residual vector writes, since several threads may be trying to write to the same location in memory. A global lock on the residual vector produces extremely poor performance due to very high contention, while atomic operations on double precision data are not available in hardware on the GPU. One possible solution is to color the elements such that no two colors share a node, and then serialize computation across colors [24]. Another possibility is gathering of the element local data on a per node basis instead of assembling from elements to nodes [29]. While the coloring approach should in theory achieve better performance, memory access pattern issues on GPUs reduce its benefit, and so we have chosen to implement the latter approach.

Examining the process of developing an optimized serial GPU code reveals many of the algorithmic design decisions that would otherwise be rather unintuitive. The first step of the process was a direct translation of our existing CPU code to run on the GPU. The natural choice of granularity for parallelization was a single GPU thread per element in the mesh. Using the CUDA environment enabled an almost

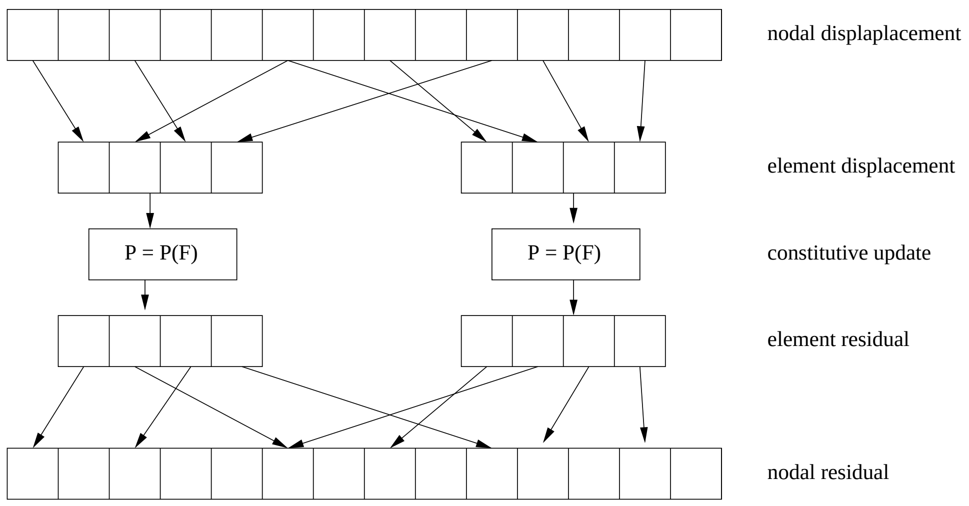


Figure 3-1: Data dependencies for residual calculation.

direct copy and paste of our C++ based application. A few changes, however, were required to conform with the limitations of the CUDA version we used. For example, the GPU we used for testing does not support C++ classes or function pointers, so choosing which element to execute or what material to use has to be done via a switch statement. Any memory allocations also have to be converted to GPU memory allocations, since a GPU thread can only access GPU memory. Generally, this only required an allocation of GPU memory via `cudaMalloc()` and a copy of the CPU structure to the GPU via `cudaMemcpy()`. However, some of our more complicated structures such as the material model implementations also contained pointers to internal allocated arrays. Since these allocation sizes were small and predictable, we chose to simply include a GPU version of these structures that statically contains the required memory.

The primary issue with this initial conversion is proper synchronization of the GPU threads. As shown in Figure 3-1, each element thread calculates a local residual vector that must then be inserted into the global residual vector. Our synchronization is done by reversing the manner of assembly, and turning it into a gather operation. We loop through the nodes in the mesh, and for each node we gather the values from each element that contains that node. Performing this operation requires a reverse connectivity map, from nodes to elements, which can be obtained directly from our

`MeshTopology` object. On the GPU, the loop through the nodes in the mesh is performed in parallel, with a thread per node. Since we are writing a value per node, there is no write contention for this process.

To test our serial code we ran on a machine with a Tesla C1060 GPU (78 GFLOPs double precision peak performance, 102 GB/s memory bandwidth) and dual quad-core Intel Xeon X5550 Nehalem processors operating at 2.67 GHz. We shall use GPU vs. CPU speedup in assembly time to measure the relative performance of our efforts and the merit of optimization strategies. As has been noted in [24] and [28], the speedup metric is highly dependent on the CPU implementation and varies with hardware, and is thus not as useful as an absolute metric of performance. A direct translation of the code with only the required modifications detailed above produced a speedup of approximately 2.5x on the GPU vs. a single CPU core. Unfortunately, any speedup of less than approximately 7x is not beneficial on this machine since we can simply run an OpenMP version of our code on the eight core CPU. Perhaps a more telling metric is that this version only reaches 3 GFLOPs of double precision performance and less than 10 GB/s memory bandwidth, which is far less than the theoretical peak performance of the GPU.

The first performance improvement comes from explicitly unrolling several of the loops in our code. Although the CUDA compiler will in general unroll loops for which it knows the bounds at compile time, we did not see this behavior for nested loops. Our code includes many matrix multiplications for relatively small matrices of known sizes, such as in the material constitutive update or in the integration of the stress field. Unrolling these loops in the GPU code provided a factor of 2 performance improvement, bringing us to a total of 5x speedup over the CPU. All further improvements require at least some knowledge of graphics hardware.

Properly understanding the memory hierarchy on GPUs and CPUs is critical to high performance, as many programs are memory bandwidth limited and not CPU limited. A schematic overview of the entire memory structure is shown in Figure 3-2, along with approximate bandwidths for the hardware on our test machine. Graphics memory is divided into fast but small shared memory and slow but large global mem-

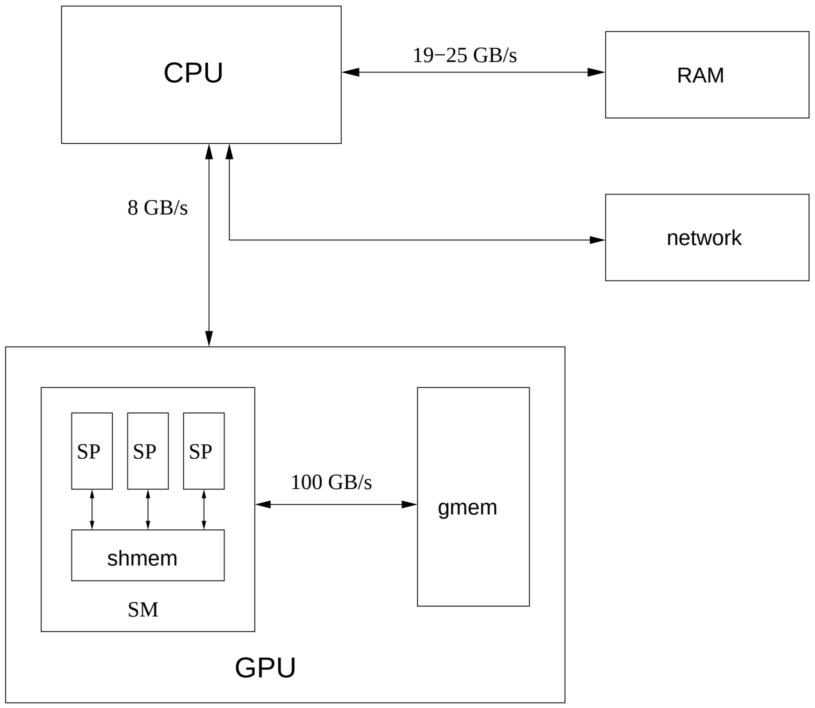


Figure 3-2: Memory structure for GPU and CPU.

ory. Access to global memory is 100 to 150 times slower than shared memory, but the shared memory space is limited to 16 KiB per SM, split among the 8 SPs in the GT200 series. Analyzing our data requirements for residual construction shows that we need 2-3 KiB of data per element, depending on the material model used. Thus, to achieve high occupancy of threads we need to use the slower global memory. One strategy to improve access times to global memory on the GPU is proper coalescing of memory access. Coalescing requires that sequential threads access sequential addresses in global memory as well as obey certain alignment restrictions [30]. Most of the data in our application is indexed by element number and dimension of the data. The CPU implementation stores data with the element number as the most significant index, this is changed to the least significant index to achieve coalescing, see Figure 3-3. Properly coalescing memory access gives us another factor of 2 performance, bringing us to a about 10x speedup over the CPU.

Our residual calculation kernel requires a high number of registers due to the complexity of the constitutive calculation, and thus achieves fairly low occupancy on the graphics card. Since the GT200 generation of graphics cards lacks an L1 cache,

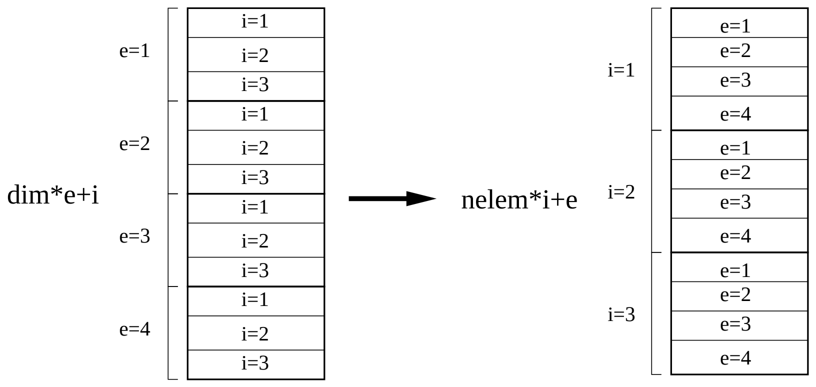


Figure 3-3: Our program is parallel across elements, which is index e. Element data can be vectors or tensors, which is index i. To coalesce memory access, we must store data with the element index, which determines the thread, as the least significant index (right side of diagram).

it uses a large number of threads and essentially free thread switching to mask the long latency on global memory access [21]. However, when a thread is switched out its context must be stored. Thus, the registers on an SM must be split between the currently running threads on the SP, as well as any sets of threads that have stalled and been swapped out. Since we cannot reduce the number of threads we are using, achieving higher occupancy with our kernel requires increasing the granularity of our parallelization. Evaluating the integrals to compute our residual force vector requires numerical integration, which we accomplish via Gaussian quadrature. Each element has a set of quadrature points, and the quantity to be integrated is evaluated at these points and the results summed with an established weighting. For example, a second order tetrahedron has four quadrature points.

We modify our kernel to have a thread per quadrature point instead of a thread per element, which is similar to the approach taken by Komatitsch et al. [24]. To evaluate the integrand, each thread requires all of the nodal data on the element, such as displacements and the residual vector at all the nodes. The quadrature point values and derivatives are then interpolated from these values. This nodal data is stored in shared memory, and the work of loading it from global memory and storing it is split among the quadrature threads. We have to be careful when assembling the residual vector to ensure that multiple threads do not attempt to write to the same location

Table 3.1: Performance of the GPU code for an example problem with a neohookean material model. GPU performance is listed in GFLOPS for residual computation and assembly.

Elements	CPU [s]	GPU [s]	Speedup	GFLOPS
160	2.375	0.875	2.71	2.59
4320	2.264	0.169	13.40	13.42
10240	2.317	0.128	18.11	17.73
43940	2.383	0.114	20.86	19.85
160000	2.410	0.114	21.08	19.84
439040	2.403	0.114	21.16	19.97

in shared memory. In the case of finite elements, each quadrature thread needs to add a value to the running sum of the residual vector for each of the ten nodes. To ensure that we do not have overlapping writes, each thread writes its partial result into a location in shared memory. Then, the threads perform reduce operations in parallel on separate values. There are 10 nodes and 3 spatial directions and thus 30 values to be reduced, which allows for high efficiency with 4 threads. Implementing this finer level of parallelization brings us to a factor of 21x speedup.

Performance results for a variety of problem sizes are listed in Table 3.1. Speedup results are reported, showing the ultimate 21x speedup for the largest mesh size. A more telling result is the GFLOPs performance. For the largest meshes, our GPU code reaches approximately 20 GFLOPS in double precision, which is about 25% of the performance capability of the graphics card. This level of arithmetic performance is fairly high for an application with such a high amount of random memory access.

3.3 CUDA + MPI Hybridization

Although running our code on a GPU produces impressive speedups over running on a CPU, we become very limited in problem size since GPU memory sizes are limited to about 4GB. Thus, we are interested in hybridizing our code to run on clusters of GPUs. This combination of shared memory algorithms on a single node and distributed memory algorithms across the cluster of nodes is one possible implementation of hybrid parallelism. There are many possible models of hybrid parallelism [31], but for this code we focus on CUDA and MPI hybridization.

One of the earliest investigations of hybrid parallel GPU computing for finite element problems was performed by Goddeke in 2007 [32]. Once again, their acceleration efforts focused on speeding up the linear solver, specifically the sparse matrix vector multiplication. However, their work demonstrated that adding even outdated GPUs to the cluster could create noticeable speed ups. In 2010, Komatitsch et. al. also accelerated their geophysics code to run on a cluster of GPUs, with promising results [33].

Before embarking on a program of hybrid parallelization, it is important to ascertain if the effort will provide worthwhile benefits. If we accelerate the computation on each node, we will reduce compute time relative to communication time for a single time step. Assuming that communication time is unaffected by our efforts, the total speedup we achieve can never be greater than $1/f_c$ where f_c is the fraction of communication time in the total program execution. For example, if communication is half of the program execution time, i.e. $f_c = 0.5$, then we can never achieve more than a factor of 2 speedup for the parallel code. Thus, we require that the communication time be small compared to the total timestep so that we can maximize our potential speedup. Figure 3-4 shows the strong scaling properties of the CPU parallel portion of our code. This benchmark reveals that we only spend approximately 2% of the timestep in communication. Therefore, our program is suitable for hybrid parallelization.

In the non-GPU version of our code, we parallelize across processors using domain decomposition for the problem and MPI for communication between nodes. The problem geometry is divided into components of equal numbers of elements by the METIS package [34]. Each processor owns a submesh of elements, as well as fragments of the overall displacement, residual and other fields. In a dynamic problem, each processor performs the predictor and corrector step independently, since these steps require only data values at a given node. As shown in Figure 3-5, nodes on the boundary between processors will see force contributions from elements on both

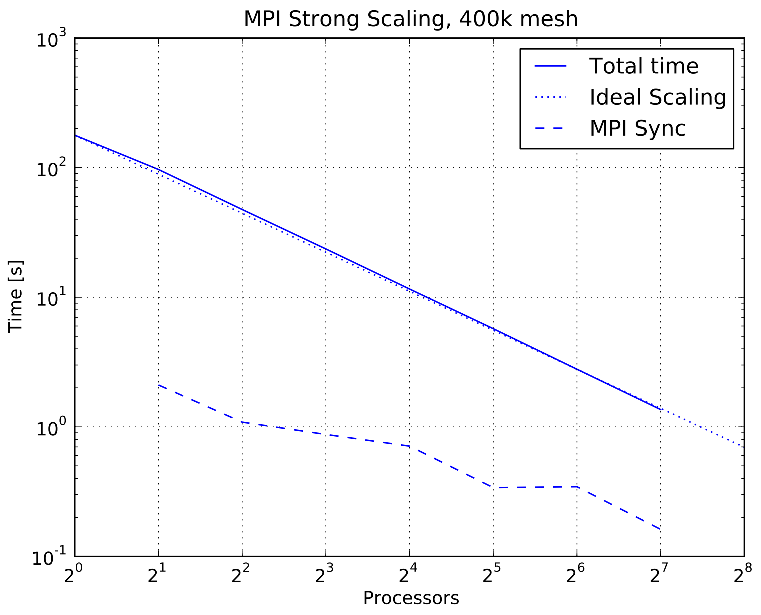


Figure 3-4: Strong scaling properties of the CPU version of our code. Time given is for 100 timesteps of an explicit problem on a 400,000 element mesh. The upper dashed line shows ideal strong scaling, while the lower dashed line shows communication time.

processors. Thus, the only communication we require is the synchronization of residual vector values of the nodes on the interprocessor boundaries. The actual residual computation is done locally on the processor, after which we perform a sum reduce operation across all of the shared nodes.

Our method lends itself to naturally to the hierarchical structure required for hybrid parallelism, as shown in Figure 3-6. After the domain decomposition, each processor still has many thousands of elements. Thus, we can apply our shared memory serial algorithm to the individual processor problem with the addition of the synchronization step described above. This synchronization step requires a transfer of data from the GPU to the CPU for broadcast across the network.

Each individual processor only communicates with its neighbors, i.e. processors whose mesh partition borders the given processor. Each processor stores a list of the processor IDs of its neighbors, as well as a communication map specifying the local indices of shared nodes. On a synchronization step, each processor loops through its neighbors, allocates a buffer the size of the communication map, fills that buffer

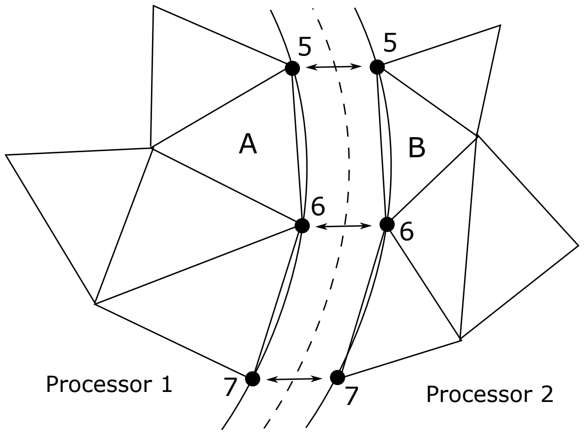


Figure 3-5: Synchronization step for distributed memory parallel algorithm. Node 6 sees contributions from both elements A and B, which are on different processors.

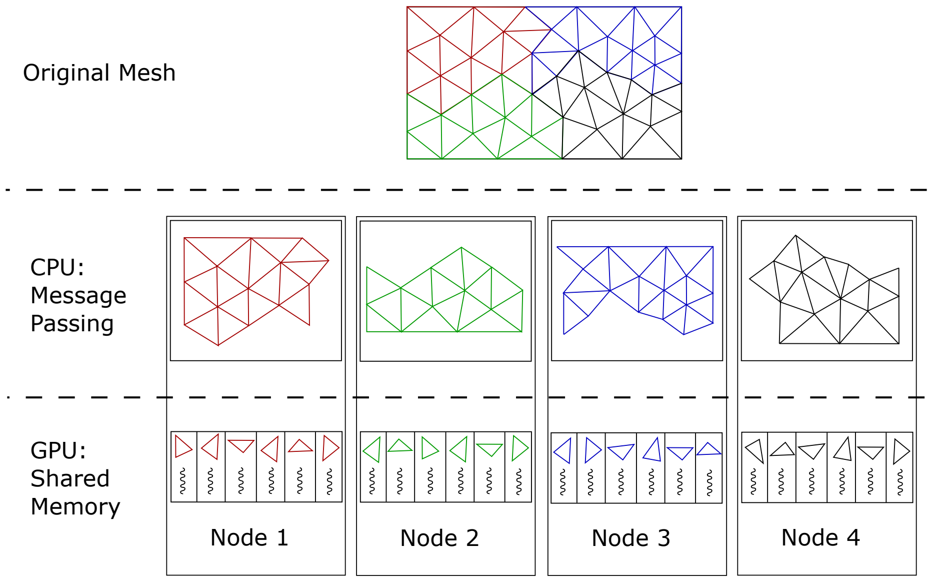


Figure 3-6: Hybrid parallel decomposition of a mesh. Each CPU has a partition of the mesh, and communicates boundary data. The attached GPU runs a single element per core, with synchronization between individual threads.

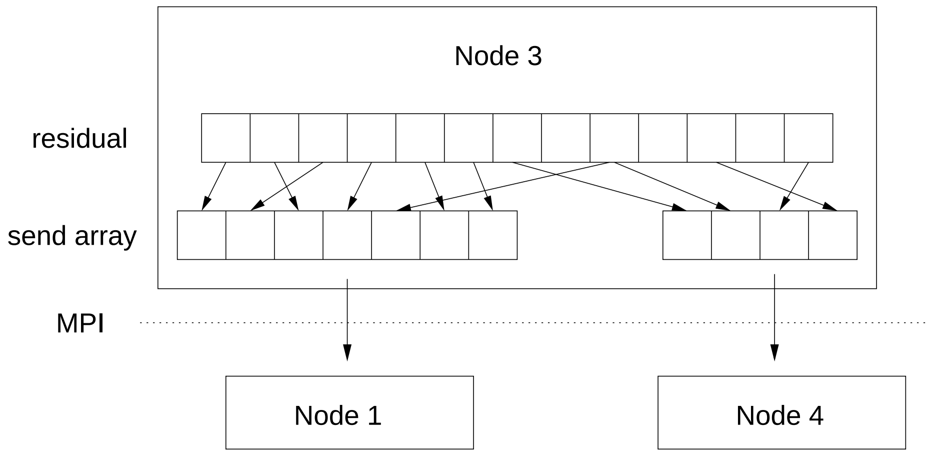


Figure 3-7: MPI communication map operation. Node 3 needs to synchronize with Nodes 1 and 2, and the communication maps specify which residual values it must pull and send across the network.

according to the value specified by the map, and sends it via an MPI non-blocking send (Figure 3-7). Each processor also posts a non-blocking receive for each neighbor, and then takes the received arrays and adds them to the nodal array being synchronized.

As a first test for hybrid parallelism, we simply copy the entire residual vector from the GPU up to the CPU. Then, we can use the existing parallel infrastructure to synchronize the array. Finally, the synchronized array is copied back down to the GPU to be used in the corrector step. For a partitioning that results in 100,000 elements per node, the residual vector is on the order of 1 MB. Even for such a small amount of data to be copied, we end up spending about 25% of the timestep in `cudaMemcpy`. This is due to the relatively small bandwidth of the PCI Express bus connecting the CPU to the GPU. As shown in Figure 3-2, the bandwidth of this connection is over an order of magnitude smaller than the bandwidth to GPU global memory. Thus, a CPU data transfer of even a 1 MB vector per timestep is an expensive operation.

Since GPU to GPU memory copies are much faster than GPU to CPU memory copies, a better approach is to copy up only the values that need to be synchronized. At initialization, the CPU communication map arrays are copied down to the GPU. To improve performance, we concatenate the arrays for all the neighbors into a single array to reduce the number of `cudaMemcpy`s required. The GPU is not concerned

Table 3.2: Relative times of hybrid synchronization steps

Part	Relative Time
CUDA gather	14%
Copy to Host	12%
MPI Sync	57%
Copy to Device	9%
CUDA scatter	8%

about the details of which neighbor each piece of information goes to, it simply knows that the CPU requires certain values from the residual vector. The synchronization step now proceeds in five parts:

1. CUDA gather - GPU loops through concatenated communication maps, assembling values from residual vector into a single array
2. Copy to Host - Copy concatenated send arrays to the CPU via `cudaMemcpy`
3. MPI Sync - Communicate buffers via non blocking sends, buffers are pointers into concatenated send array
4. Copy to Device - Copy concatenated array of receive buffers to the GPU via `cudaMemcpy`
5. CUDA scatter - Once again, GPU loops through communication map array and adds values into residual vector

The relative time required for each part for a typical timestep is shown in Table 3.2. The extra overhead for the GPU copies is approximately equal to the MPI communication time. Note also that the assembly of the communication buffers is also now done in parallel on the GPU, exploiting the higher bandwidth of the GPU main memory over CPU main memory.

Using this synchronization algorithm, we proceed to measure the strong scaling performance of our GPU code, shown in Figure 3-8. This plot also shows results for the CPU code, and both exhibit near perfect strong scaling. An unforeseen benefit of

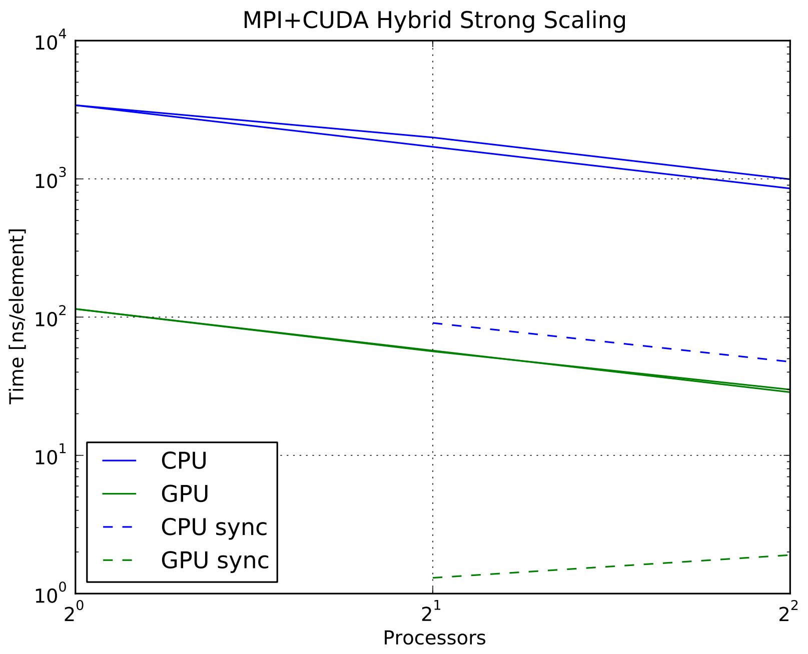


Figure 3-8: Strong scaling for MPI+CUDA code. Nodes have Xeon E5530 processors, a single C1060 and IB QDR interconnect

the hybrid code is that the synchronization time is actually faster than the pure CPU code. Further profiling of the CPU code reveals that much of the synchronization time is spent waiting in `MPI_Barrier`, due to small load imbalances and memory access timing differences between processors. As we accelerate the serial code, the relative size of these imbalances remains the same, but the absolute time measure of the imbalance decreases. Thus, the absolute measure of synchronization time actually decreases as we accelerate computation, meaning that our communication time factor f_c changes. The implication is that our speedup can be even greater than the original maximum estimate of $1/f_c$.

3.4 Seamless integration of GPGPU acceleration

GPU acceleration is only truly useful if it is seamlessly integrated into a code. The application writer should not have to be concerned about the details of the GPU implementation, and ideally no extra steps should be required to run a program on the GPU. One of the driving goals of the design decisions detailed in Chapter 2 was seamless integration of GPU computing in our code.

The element set structure described in Chapter 2 greatly simplifies the integration of GPU computing into our code. We create a new element set called `ElementSetTet2GPU` that encapsulates the serial FEM code detailed earlier in this chapter. Since an element set represents an entire collection of elements, we can include the somewhat delicate assembly algorithm directly in the element set code. This reduces the complexity of the overall structure of the code, and minimizes the intrusiveness of GPU modifications.

Our nodal and quadrature fields also need to be modified so that memory allocations occur on the GPU. Once again, we can minimize intrusiveness by creating specialized GPU objects, `NodalFieldGPU` and `QuadratureFieldGPU`. The GPU version of the elemnt set can then access the GPU data memory pointers from these structures for use in the residual assembly kernel. One difficulty is that we would like to maintain the same interfaces for CPU code and GPU code. Our goal is that an application writer can simply select to enable or disable GPU acceleration at the beginning of the application, and all of the rest of the code remains the same. We accomplish this goal through the use of private implementation objects. For example, when GPU compilation is enabled the `NodalField` class is simply a thin layer that contains the usual `NodalField` interface and a pointer to an implementation class. This implementation class is either a `NodalFieldGPU` or a `NodalFieldCPU` object, depending on whether or not GPU acceleration is enabled. Any method calls on the nodal field get delegated to one of these objects. This structure allows us to retain the same interface to the element sets for the GPU code and the CPU code, and encapsulates all GPU relevant code within one object.

Another change required is a similar delegation scheme with our solvers. The systems can remain unchanged, since they are simply containers for the fields. For example, with our explicit newmark solver we again retain the same interface in the `ExplicitNewmarkIntegrator`. This class now also contains a pointer to an implementation class, which can be either a CPU integrator or a GPU based integrator, with the appropriate class instantiated based on whether or not GPU acceleration is enabled. The CPU class remains as before, while the GPU class simply pulls the GPU memory pointers from the GPU implementations of the `NodalField`s and calls the appropriate kernels for the predictor and corrector step.

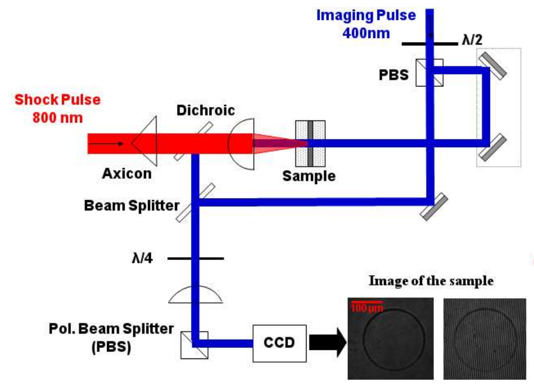
One additional component is necessary to complete the integration of GPU acceleration. We require a singleton object to handle the initialization of CUDA and also to offer an interface for enabling or disabling GPU acceleration. The constructor for this class enables CUDA via the appropriate library calls and selects the best GPU to use out of the available options. The destructor calls any clean up functions for CUDA and our interface. With this interface, an application writer can enable GPU acceleration through one function call. All the remaining code in the application can stay the same. Data is only copied between the GPU and CPU for the initialization of boundary conditions at the beginning of the problem, and for any output. Further performance could be achieved by using asynchronous memory copies for moving output data up to the CPU and a separate thread for output, since this can occur in parallel with the calculations. However, this optimization has not yet been put into place.

Chapter 4

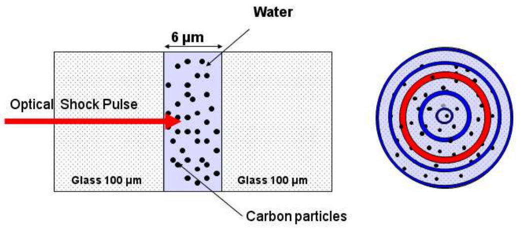
Numerical Tests and Application Problems

The correctness of our code is established by a combination of unit tests and full simulations of simple problems. These tests are run nightly as part of a nightly build and test suite using the CTest framework. Verification tests include 2D and 3D patch tests for the elements, including static and dynamic tests corresponding to uniform strain and strain rate fields, respectively. In addition, simple linear problems with known solutions such as a plate with hole subject to a uniform remote stress are included to verify the overall functionality of the code and material model.

To demonstrate the applicability of our new finite element framework, we present several example problems that highlight some of the new features. The first problem corresponds to the propagation of converging as well as diverging cylindrical shock waves generated by a cylindrical picosecond laser pulse in a water film. The second problem corresponds to wave propagation in a micro-truss, and shows the parallel capability of the code. Finally, we demonstrate the multiphysics capability via a superelastic tension problem.



(a) Laser and mirror setup



(b) Shocking geometry on a sample

Figure 4-1: Overview of laser induced shock experimental setup.

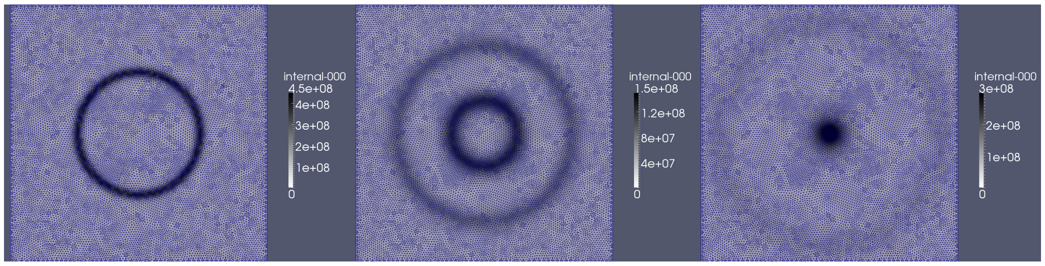
4.1 Laser induced shock

A recently developed experimental technique involving a laser induced shock shows promise for novel experimental inquiry into material behavior at extreme stresses while providing micromechanical response details [35]. In this technique, a laser beam is focused on a concentric, ring shaped region of the material of interest via a set of mirrors and beam splitters (Figure 4-1a). The energy from the 300 picosecond laser pulse causes a rise in temperature in the region and ultimately a high pressure zone. This cylindrical high pressure zone induces two shocks - one propagating inwards and one propagating outwards (Figure 4-1b). The inwards propagating shock ultimately focuses at the center of the ring, creating a spot of very high pressure. Unfortunately, it is very difficult to experimentally measure the exact pressure distribution in the material. Thus, we are interested in numerically modeling this shock propagation to better understand and quantitatively interpret the experimental results.

The specific experiment we model involves shock propagation in a water sample, with the high pressure coming from heating of carbon particles suspended in the water. The specific geometry we use is a rectangular prism, 400μm by 400μm by 6μm. In the experiment, the sample is confined between two layers of glass. For modeling purposes, we assume the glass to be infinitely rigid and allow no outwards displacement on the boundary nodes. Two simulation meshes were used - one with 214,000 elements for test runs and one with 1.7 million elements for calculations.

Table 4.1: Parameters for water material model

Property	Value
Density (kg/m^3)	$\rho = 998$
Dynamic Viscosity (Pa·s)	$\mu = 1.002 \cdot 10^{-3}$
1st Tait Parameter	$\Gamma_0 = 6.15$
2nd Tait Parameter (MPa)	$B = 304.2$



(a) Initial Mesh and Pressure (b) Shocks at 23 ns (c) Shocks at 49 ns

Figure 4-2: Coarse mesh used for investigating laser induced shock propagation.

The water is modeled as a Newtonian fluid with the Tait equation of state for the pressure response and artificial viscosity for shock capturing. The Tait equation of state provides a relationship between deformation and fluid pressure:

$$p(J) = B\left[\left(\frac{1}{J}\right)^{\Gamma_0+1} - 1\right] + p_0 \tag{4.1}$$

where Γ_0 is the Tait parameter, B is a second Tait parameter, p_0 is the reference pressure of the water and J is the usual determinant of the deformation gradient. The model parameters used in this simulation are shown in Table 4.1.

The ringed laser pulse excitation is modeled as an initial ring of high pressure in the water sample, shown in Figure 4-2a for a coarse mesh. The pressure profile has a Gaussian distribution in the radial direction, centered at 95 μm with a half width of 10 μm. This initial pressure is established by applying an eigenstrain to the material. This eigenstrain creates a non identity initial deformation gradient and determinant J, which produces an initial pressure via (4.1). An updated Lagrangian scheme is then employed to maintain this strain as an initial condition. Specifically, we modify our calculated deformation gradient:

$$F = \nabla\phi \cdot F^r \tag{4.2}$$

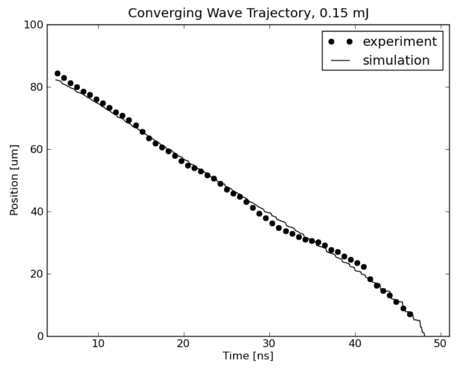
where F^r is the reference deformation gradient. In addition, we must modify our residual force calculation:

$$r_{ia} = \int_\Omega P_{iJ} N_{a,\bar{J}} F^r_{\bar{J}J} (J^r)^{-1} d\Omega \tag{4.3}$$

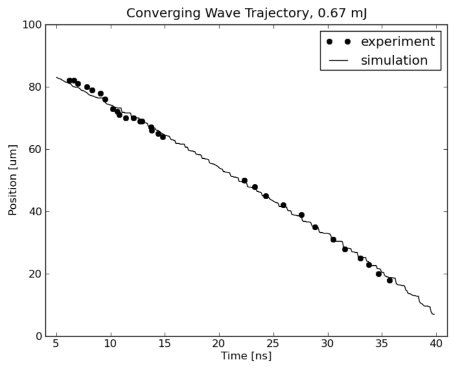
where P_{iJ} is the Piola-Kirchhoff stress tensor and $N_{a,\bar{J}}$ are the shape functions on the reference element (in the updated configuration).

The continuum equations are integrated by the explicit Newmark method described in the previous chapter. The simulation is continued until shortly after the converging wave reflects from the center, which can be 45-60 ns depending on the shock speed (Figures 4-2b and 4-2c). An automated Python-based post processing framework outside of our code analyzes the pressure data and detects the shock wave peak location to compute trajectories and wave velocities. In the simulation, we set an initial condition by establishing the initial pressure field of the water sample. In the experiment, however, the initial condition is the total energy of the laser pulse. Since the mechanism of transfer from laser energy to water pressure is beyond the scope of our modeling, we assume a linear relationship between initial pressure and laser energy based on dimensional considerations. Since zero energy corresponds to zero pressure, we only have to correlate one set of values to determine the relationship. This correlation is done by matching experimental shock trajectories with the computed simulation trajectories. For greater accuracy, we match more than one set of trajectories, these are shown in Figure 4-3.

With these calibrations complete, we proceed to verify the accuracy of our simulation against experimental results. This verification is done by comparing our computed converging shock speeds at a variety of laser energies to the experimentally determined shock speeds. Since the converging shock accelerates towards the center of the domain, we use a secant velocity between 0 and 20 ns. This comparison is



(a) Trajectory at 0.15 mJ



(b) Trajectory at 0.67 mJ

Figure 4-3: Comparison between numerically computed and experimental shock trajectories

shown in Figure 4-4. We observe that the numerical values for the converging wave match very well with the experimental values, giving us confidence in the validity of our simulation results.

With our simulation calibrated and validated against experimental results, we can proceed to examine the pressure profiles of the converging shock. This data cannot be determined through experimental measurements, and thus is the primary interest of our simulation. A plot of the pressure profiles for a 2.15mJ laser pulse is shown in Figure 4-5. This plot shows the dramatic increase in pressure at the center of the converging shock, as well as the steepening of the converging wave as it approaches the center.

4.2 Wave propagation in a micro-truss array

There has been an increased interest in protective materials applications to explore ways in which stress waves can be managed in such a way as to achieve desired mitigation metrics. These waves carry momentum and energy from impactors and blasts, both of which must be considered and managed. Momentum transfer cannot be reduced, but we can extend its time scale, thus reducing force. Energy transfer can be reduced, by using a material that can absorb energy in plastic deformation.

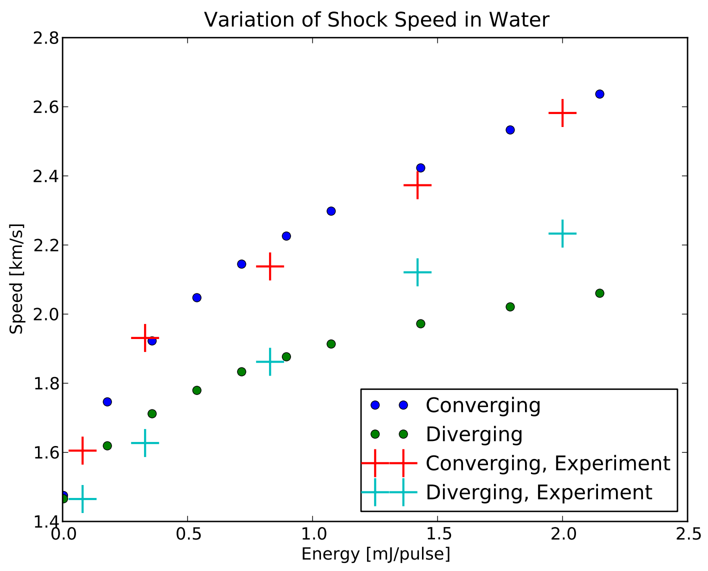


Figure 4-4: Comparison between numerically computed shock speed and experimental shock speed for a variety of initial energies.

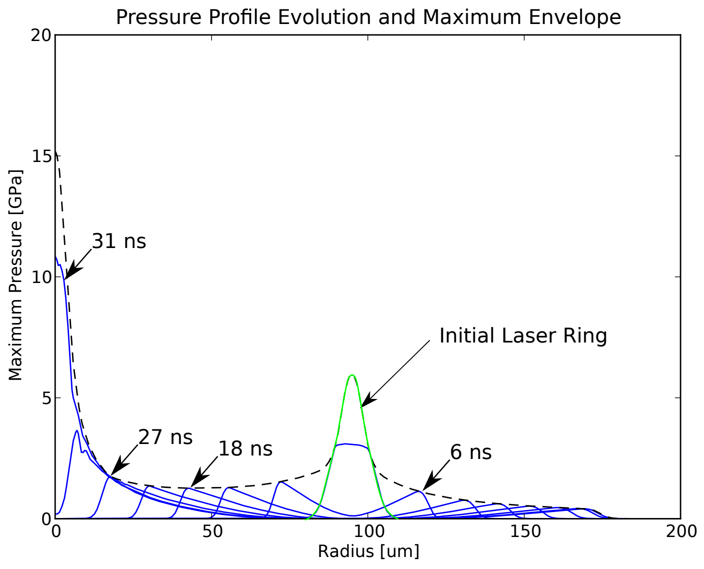


Figure 4-5: Pressure profiles for converging and diverging shock propagations from a 2.15mJ laser pulse

Micro-truss arrays have been studied experimentally as candidates in armor design due to their excellent energy absorption properties and high structural efficiency. In addition, the voids in the array can be used for other functions, such as running a cooling liquid for heat transfer [36]. Finally, the regular, periodic structure of micro-truss arrays can be exploited to direct the propagation directions of stress waves.

Numerical analysis of micro-truss arrays has so far concentrated mainly on the plastic dissipation of energy under compressive loading [37] using quasi-static analysis. The periodic nature of micro-truss arrays implies that they will have interesting wave propagation properties. Ruzenne and Scarpa investigated wave propagation in a 2-D hexagonal array using beam elements and eigenvalue analysis [38]. The arrays showed strong directionality for wave propagation, as well as band-gap frequencies. We are interested in studying the wave propagation properties for a 3-D micro-truss array using a full continuum finite element model for the array as opposed to beam elements. This enables the description of potentially defining features of the response such as scattering of longitudinal and shear waves at the truss nodes, etc.

4.2.1 Mesh generation

To allow for parametric studies, it is desirable to be able to automatically generate micro-truss array meshes. One possibility is to generate an entire mesh for an arbitrary number of units in each dimension. However, due to the periodic structure of the truss, it is possible to simply generate the mesh for a single unit and then tile this unit mesh to obtain our total array. This hybrid structured-unstructured mesh has clear advantages in terms of disk usage. In addition, we can load an entire array in parallel, with each processor loading a unit mesh and the communication maps statically determined at the time of mesh creation. This approach allows us to scale simulations to the size of the available cluster without the limitation of loading a mesh on one processor first. For simplicity, we model a pyramidal micro-truss array so that truss units tile as cubes.

To ensure a conforming mesh, we must make sure in the generation of a single mesh unit that the tetrahedron faces present on one side of the mesh match the faces

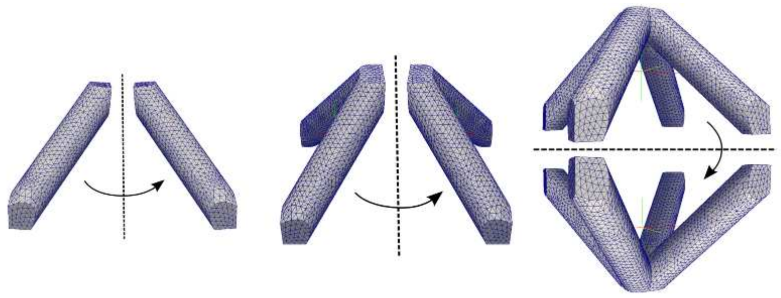


Figure 4-6: Assembly of microtruss mesh unit from individual leg mesh.

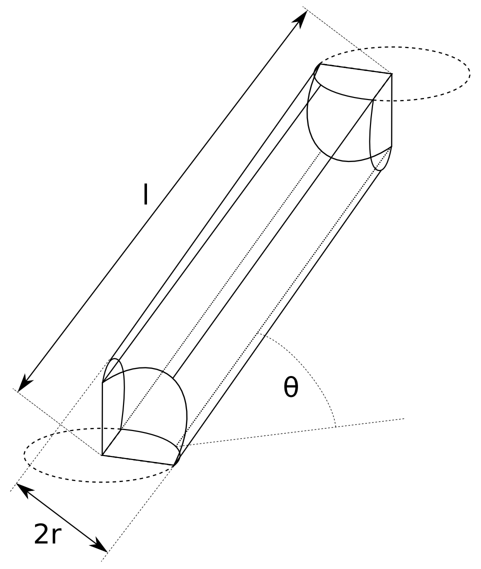


Figure 4-7: Boundary representation for a truss leg with geometric parameters.

on the other side. To satisfy this requirement, we generate a mesh for a single leg and then assemble it as shown in Figure 4-6 to create the octahedral mesh unit. This assembly process creates an eight-fold symmetry in the mesh that guarantees opposing faces of the unit will have matching nodes. To automate the process, we create the individual leg mesh from a computer generated BRep description. The BRep is created by a python script, whose input parameters are the length of the leg, its radius, and the truss angle. The geometry of this BRep is shown in Figure 4-7, along with the 3 dimensional parameters: r, l, and θ.

The complete process for creating a micro truss mesh unit is then:

1. Automatically generate a BRep for a truss leg from input parameters r, l and θ

2. Create the leg mesh from the BRep with the desired number of elements through

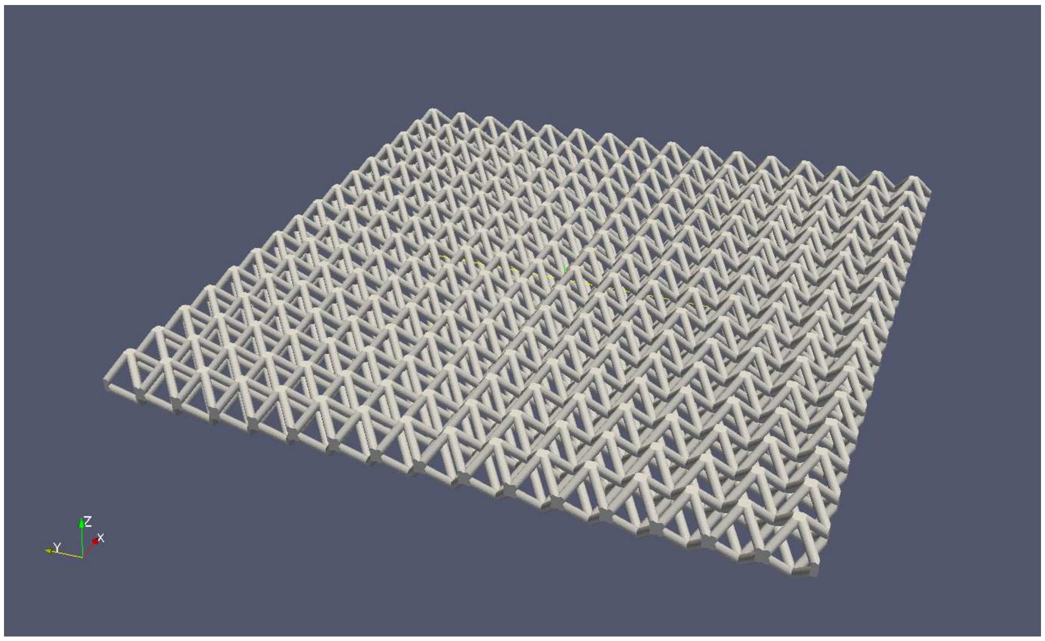


Figure 4-8: Scaling test for micro truss simulation on 256 processors, with 12 million elements.

any external meshing tool

3. Flip the mesh and stitch the resulting overlapping nodes for each spatial axis x,y and z
4. Create static communication maps by matching nodes on opposing faces

This process is entirely automated via a script that takes as input the 3 leg parameters and a mesh size parameter. A single unit is created from this script, and the actual micro-truss array mesh is then created in parallel by tiling a single unit per processor. Further details on the geometry of the BRep and the actual code for the generating script can be found in Appendix A.

4.2.2 Wave propagation analysis

To demonstrate the scalability of our mesh loading, we run a test case of a planar 16x16x1 micro-truss array on 256 processors. Each processor loads an individual 50,000 element unit mesh, for a total mesh size of 12 million elements. A view of the entire mesh is shown in Figure 4-8, showing that the mesh was loaded correctly

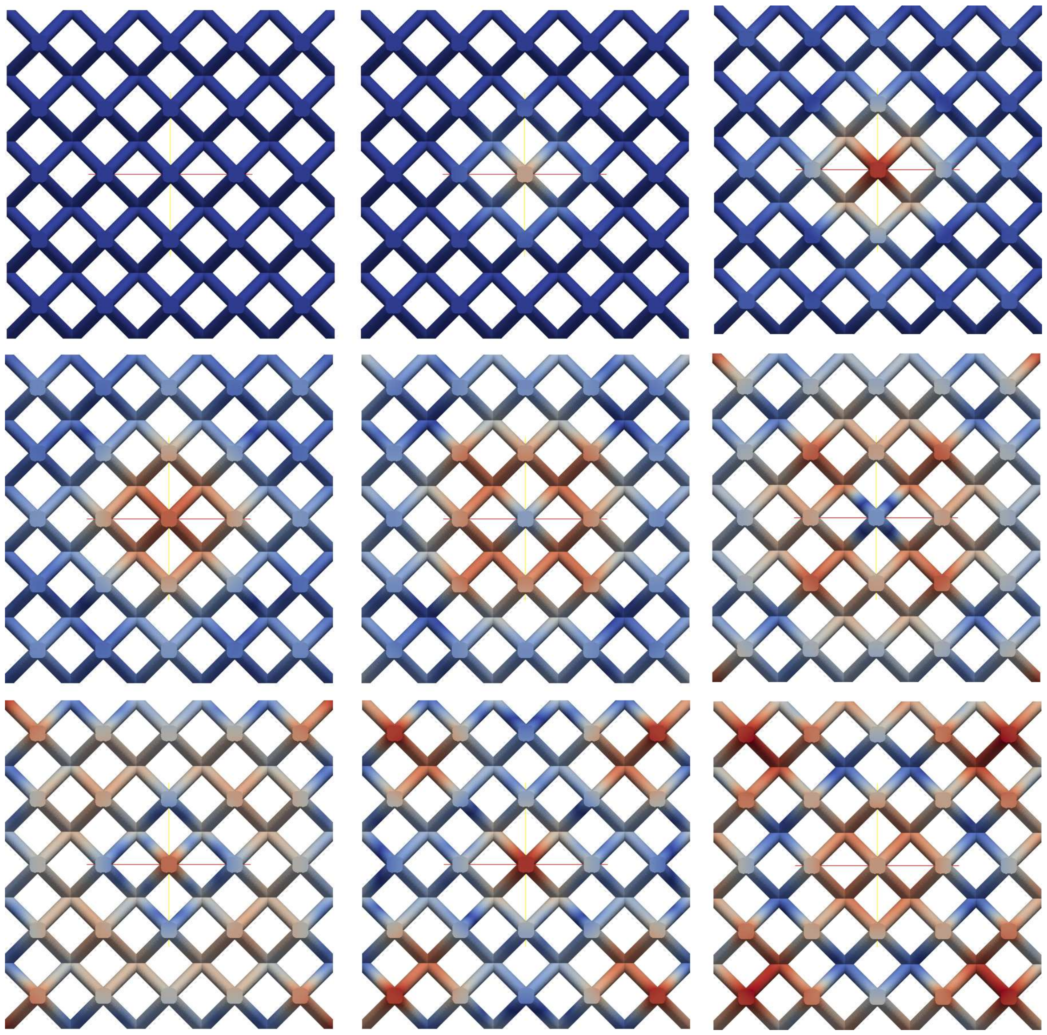


Figure 4-9: Top view of wave propagation in 5x5 micro-truss array. Colors indicate displacement.

with the correct spatial offset on each processor. To verify that the interprocessor communication maps have been correctly constructed, we then apply an impulse at the center of this mesh and observe the propagation of the wave. The purpose of this test is not to observe the nature of the propagation or to make any quantitative measurements. Instead, we want to determine that the waves smoothly progress across processor boundaries to verify that our parallel mesh loading algorithm works correctly. Visual inspection of the resulting data files confirms that the wave generated by the impulse propagates across the mesh correctly.

Our primary interest is the observation of wave propagation through the periodic

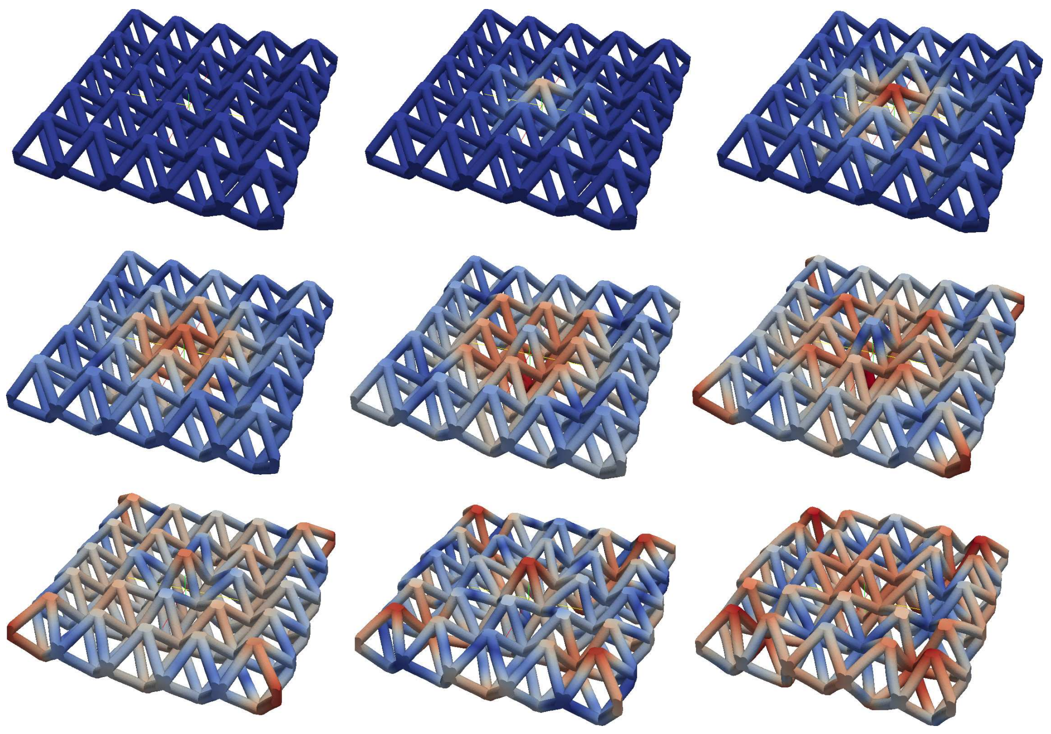


Figure 4-10: Side view of wave propagation in 5x5 micro-truss array. Colors indicate displacement and the mesh is deformed by displacements scaled 20 times.

Table 4.2: Parameters for micro-truss material

Property	Value
Density (kg/m^3)	$\rho = 7000$
Young's Modulus (GPa)	$E = 211$
Poisson Ratio	$\nu = 0.3$

structure of the micro-truss array. Thus, our next test is a simple wave propagation problem. Our array is again planar, with 5 units in the x and y directions. The truss legs are 1 mm in length, with a radius of 0.1 mm and a leg angle of 45 degrees. Once again, each mesh unit consists of 50,000 individual elements. For the truss material, we employ a neohookean model with parameters as shown in Table 4.2, which correspond to steel. The boundary conditions are force free everywhere, with an applied displacement on the top of the center truss unit. The displacement is periodic in time, with an amplitude of 0.01 mm and a frequency of 100,000 rad/s. The extremely high frequency is so that we can observe several oscillations of the displacement forcing given the restricted time size step with an explicit integrator. The resulting wave propagation is shown from the top in Figure 4-9 and from the side with exaggerated deformations in Figure 4-10.

We can clearly see the directional nature of the wave propagation in the micro-truss array. Namely, at a given distance the wave amplitude appears to be highest on the four corners of a square centered about the forcing point. The displacement along the edges of this squares is much less than the corners. This indicates the presence of preferred directions of propagation, which likely have to do with the four-fold symmetry of the octahedral truss unit when viewed from above. Further analysis of the properties of these micro-truss arrays could include eigenvalue analysis of the mesh to determine its primary vibration modes, which would allow a full analysis of its wave propagation properties.

4.3 Superelastic bar under tensile load

Our final test demonstrates the multiphysics capability of our code. We consider a bar constructed from a shape memory alloy, such as single crystal Cu-Al-Ni. Application of a uniaxial tensile load will induce a phase transformation between the austenitic and martensitic phases of the crystal. The so called superelastic effect refers to the significant strains developed in this phase transformation, which are fully recoverable upon unloading. We follow the formulation in [39], which introduces an energetic length scale, l_e and a dissipative length scale l_d into the free energy and dissipation rate, respectively. For simplicity, we shall only consider the energetic length scale in this problem.

4.3.1 Material model

Our model uses separate fields for the deformation field φ and the volume fraction of martensite ξ. We define the usual deformation gradient $\mathbf{F} = \nabla_0 \varphi$ and the Green-Lagrange strain tensor $\mathbf{E} = \frac{1}{2}(\mathbf{F}^T\mathbf{F} - \mathbf{1})$. Assuming small strains, we can decompose the strain tensor as $\mathbf{E} = \mathbf{E}^e + \mathbf{E}^t$, where $\mathbf{E}^e$ is the elastic component and $\mathbf{E}^t$ is the phase transformation part. We can now define the free energy per unit volume as

$$\psi(\mathbf{E}^e, \xi, \nabla_0 \xi) = \frac{1}{2}\mathbf{E}^e : \mathscr{C} : \mathbf{E}^e + \frac{\lambda_T}{\theta_T}(\theta - \theta_T)\xi + \frac{1}{2}S_0 l_e^2 ||\nabla_0 \xi||^2 \tag{4.4}$$

where $\mathscr{C} = \mathscr{C}(\xi)$ is the elastic moduli of the mixture of austenite and martensite, θ_T is the equilibrium temperature between the two-phases in a stress free state, λ_T is the latent heat, and S_0 is a model parameter. The final term of this free energy introduces a non-local term with the energetic length scale l_e.

We assume the usual linear elastic material response to the elastic strain $\mathbf{E}^e = \mathbf{E} - \mathbf{E}^t$

$$\mathbf{S} = \mathscr{C} : (\mathbf{E} - \mathbf{E}^t) \tag{4.5}$$

where $\mathbf{S}$ is the second Piola-Kirchhoff stress tensor. In general, the inelestic strain is defined as $\dot{\mathbf{E}}^t = \dot{\xi}\Lambda$ where Λ is determined via a flow rule. For simplicity, in this initial test problem we will consider an inelastic strain defined by

$$\mathbf{E}^t = \xi\Lambda = \xi\epsilon_t\sqrt{\frac{3}{2}}\frac{\mathbf{S}^{\text{dev}}}{||\mathbf{S}^{\text{dev}}||} \tag{4.6}$$

where ϵ_t is a material parameter for the maximum transformation strain and $\mathbf{S}^{\text{dev}}$ is the deviatoric component of the second Piola-Kirchhoff stress tensor. Our mechanics system uses the usual force balance

$$\nabla_0 \mathbf{P} = 0 \tag{4.7}$$

where $\mathbf{P} = \mathbf{FS}$ is the first Piola-Kirchhoff stress tensor.

The introduction of the volume fraction in the free energy (4.4) with a gradient term requires the solution of a second, coupled system for the volume fraction component. In this simplified problem, we ignore any terms involving time derivatives of the volume fraction, $\dot{\xi}$. We also assume a single set of elastic moduli for the material, so $\mathscr{C}_{,\xi} = 0$. These assumptions produce the volume fraction force balance

$$\mathbf{S} : \Lambda + S_0 l_e^2 \nabla^2 \xi = Y + B_t \tag{4.8}$$

where Y and B_t are material parameters describing the resistance to phase transformation and thermal back stress, respectively.

4.3.2 Implementation of the coupled system

As stated in the introduction, one of the key contributions of the framework developed in this thesis is its ability to simulate multiphysics problems. In principle, we can solve an arbitrary number of coupled PDEs where each problem is represented by a system object. For the superelastic problem, we need to solve the balance equations (4.7) and (4.8), and so our unknown variables are the displacement field, u and the volume fraction field ξ. The mechanics problems couples to the volume fraction problem through the second Piola-Kirchhoff stress in (4.8) while the volume fraction problem couples to the mechanics problem through the inelastic strain computed in (4.6). A staggered solver shall be employed to solve the coupled problem, which shall solve each PDE in turn, transferring the required fields, until the solution fields converge.

Within the context of our finite element framework as described in Chapter 2, we use a standard `StaticsSystem` for the mechanics component of the problem and a new `VFSystem` for the volume fraction component. Both of these classes derive from

Table 4.3: Parameters for shape memory alloy material

Property	Value
Density (kg/m^3)	$\rho = 7000$
Young's Modulus (MPa)	$E = 10000$
Poisson Ratio	$\nu = 0.0$
Maximum Phase Transformation	$\epsilon_t = 0.04$
Thermal Back Stress (MPa)	$B_t = 4.0$
Volume Fraction Gradient Parameter (MPa)	$S_0 = 100$
Resistance to Phase Transformation (MPa)	$Y = 1.0$
Energetic Length Scale (m)	$l_e = 0.1$

the `NonlinearSystem` base to allow them to interface with our existing non-linear solvers. We then add a new coupled solver class to perform the iterative staggered solve using Newton-Raphson solvers for the individual systems. The mechanics problem is solved first, after which we must transfer the second Piola-Kirchhoff stress to the volume fraction system. The transfer is accomplished through the `Transfer` method in the abstract `System` class, which allows for any given nodal or quadrature field to be transferred into any other nodal or quadrature field in the `System`, with the assumption that both fields are defined on the same `FunctionSpace`. Transfers from a nodal field to a nodal field or a quadrature field to a quadrature field are straightforward copies. The mixed transfers require the use of the `Interpolate` and `Extrapolate` methods provided by the function space class. After the stress transfer, we solve the volume fraction problem. Finally, the inelastic strain is transferred to the mechanics system, which requires interpolation of the volume fraction field ξ to the quadrature points.

4.3.3 Results for static loading

We demonstrate the capabilities of the coupled solver with a static problem of tensile loading on a bar. This problem was already solved in 1D in [39], but it is useful to use it here as a test case for the 3D code. Our mesh geometry is a bar with dimensions 100 cm by 20 cm by 20 cm, fully restrained on one end and with an applied displacement on the other end. The volume fraction boundary condition is $\xi = 0$ at both ends of the bar, with the assumption that the mechanical boundaries inhibit the phase transition. For this first investigation, we use a relatively coarse mesh of 200 elements with second order tetrahedra. The material properties of the shape memory alloy are shown in Table 4.3.

For this simple static case, we investigate the development of the phase transition under increased loading to demonstrate the correct coupling of the problems. We increase our applied displacement from 1 cm up to 3.5 cm, and observe the change in the volume fraction of martensite in the bar. The results of this test are shown in Figure 4-11. In this simulation we can clearly see that the amount of material undergoing the phase transition increases as we increase our load. Under the 3.5cm displacement, we see that the entire center of the bar has transitioned to martensite. The development of the volume fraction in the material verifies that our coupling code is correctly transferring fields between the two problems. Furthermore, the staggered solver converges within 2 iterations in each load case, where our convergence criteria is the vanishing of the volume fraction increment between each staggered iteration. We use an absolute tolerance of 8×10^8 and a relative tolerance of 1×10^8, compared to the increment in the first iteration. The fast convergence we observe is unsurprising as our deformations are essentially within the linear regime of the material.

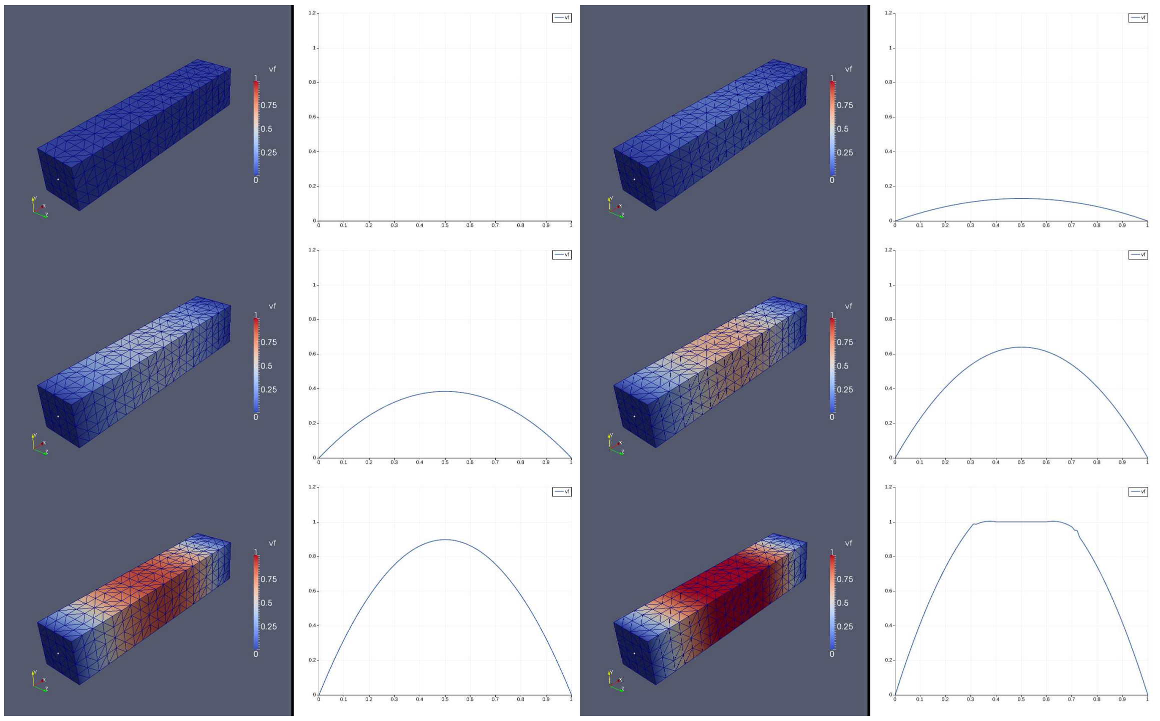


Figure 4-11: Evolution of martensite volume fraction along the length of the bar under increased displacement loading. Displacements are from left to right, top down: 1cm, 1.5cm, 2cm, 2.5cm, 3cm, 3.5

Chapter 5

Summary and Conclusions

In this thesis we have presented a new, object-oriented, hybrid parallel computational framework for solid mechanics and PDEs in general. An overview was provided of existing finite element codes and their various strategies for translating the mathematical concepts of the numerical method into software. With the lessons from these codes in mind, we presented a high level view of the modules and classes within our framework, including a description of the interface and implementation details. To accelerate our code, we translated critical numerical algorithms such as element assembly and time integration to run on GPUs and take advantage of the massive arithmetic compute power of modern GPUs. A fully hybrid approach was taken, allowing our code to run on clusters of GPUs with a mix of message passing between nodes and shared memory parallelism within nodes. Finally, our code was tested on a series of numerical example problems. These included a laser induced converging shock in water, wave propagation in micro-truss arrays with very large mesh sizes, and a coupled set of PDEs for a superelastic material.

The primary result, however, is the new framework that can now be applied to a wide variety of new problems. The main contributions of this new framework are:

- It is designed from the ground up in an object-oriented language to be extensible and to have strong encapsulation. The framework structure makes it relatively easy to add new linear or non-linear solvers, time integrators, physical systems, elements, etc. without worrying about breaking existing code. In addition, the element set structure helps ensure high performance in the critical sections of our code.

- Native support is provided for GPU acceleration on both single desktop workstations and GPU clusters. Enabling GPU acceleration for an application is a seamless process involving a single API call. The prevalence of GPUs in modern computers and the ease of enabling GPU acceleration in our framework will allow application writers to perform much compute intensive problems without having to gain access to an entire cluster.

- Our element code is written without any assumptions about the equation or equations to be discretized and solved. Combined with the encapsulation offered by our object-oriented structure, this allows our code to solve coupled multiphysics problems with an arbitrary number of PDEs. Built in methods provide the capability to transfer fields from one system to another, allowing for the easy creation of staggered solvers.

There are many avenues for further development and research with our new framework. In terms of code development, new classes can be added to simplify the addition of boundary conditions. Currently, boundary conditions are set by passing vectors to the solver class describing the type of boundary condition and the forcing vector. A better solution would be to encapsulate these with a new `BoundaryCondition` class, which would then allow us to derive classes for Dirichlet, Neumann, mixed, etc. conditions. Another improvement that can be made is the use of existing GPU libraries to provide a more STL like interface to arrays on the GPU. Our current strategy is to allow modification only on CPU vectors, which are then copied entirely onto the GPU for calculation. In terms of research, much more work can be done with the micro-truss arrays to explore the exact shape of vibrational modes as well as the possibility of band-gap frequencies in wave propagation. Our current implementation of the superelastic material uses a simplified, static version of the equations, and so a future extension is the addition of the dynamic terms and the exploration of their effects on loading and unloading. Finally, our framework provides a good interface to investigate the open question of the effectiveness of iterative linear solvers for complex material behavior. Traditionally, solid mechanics has tended to use direct solvers in implicit problems due to the poor convergence of iterative solvers for problems involving buckling, softening, or plasticity. The facilities our code provides for selecting new solvers for a problem allows for the investigation of these issues with a wide variety of serial and parallel direct and iterative solvers.

Appendix A

Generation of BRep for micro-truss

The Python script for generating a BRep for the truss leg from Figure 4-7 is shown below. The BRep format is the format accepted by the `Gmsh` mesher [40], which we use for its batch capability, simple interface and performance. The labeling for the points in the BRep as well as the choice of coordinate axes is shown in Figure A-1.

```
# create the Gmsh compatible BRep geometry for a truss leg
# @param filename name of the file for output
# @param theta the angle of the leg
# @param l the length of the leg
# @param r the radius of the leg
def gen_geo(filename, theta, l, r):
    out = open(filename, 'w')
    st = math.sin(theta)
    ct = math.cos(theta)

    L = l*math.cos(theta)
    H = l*math.sin(theta)
    rs = r/math.sin(theta)
    rc = r/math.cos(theta)
    r45 = rs*r/math.sqrt(0.5*(rs*rs+r*r))

    al = 0.5*math.sqrt(st**4+2*st*st+1+ct*ct*st*st+ct*ct)
    ra=r/(math.sqrt(2)/2*ct)

    # left end of bar points
    out.write('Point(1)={0,0,0};\n')
    out.write('Point(2)={%f,%f,0};\n'%(r45*math.cos(math.pi/4),r45*
       math.sin(math.pi/4)))
    out.write('Point(3)={0,%f,0};\n'%(rs))
    out.write('Point(4)={%f,%f,0};\n'%(-r45*math.cos(math.pi/4),r45*
       math.sin(math.pi/4)))
    out.write('Point(5)={0,0,%f};\n'%(rc))
    out.write('Point(6)={%f,%f,%f};\n'%(ra*0.5*ct/al,ra*0.5*ct/al,ra
   *0.5*(st**3+st+ct*ct*st)/al))
out.write('Point(7)={%f,%f,%f};\n'%(-ra*0.5*ct/al,ra*0.5*ct/al,
   ra*0.5*(st**3+st+ct*ct*st)/al))

# left end of bar lines
out.write('Ellipse(1)={2, 1, 3, 3};\n')
out.write('Ellipse(2)={4, 1, 3, 3};\n')
out.write('Ellipse(3)={2, 1, 6, 6};\n')
out.write('Ellipse(4)={5, 1, 6, 6};\n')
out.write('Ellipse(5)={4, 1, 7, 7};\n')
out.write('Ellipse(6)={5, 1, 7, 7};\n')
out.write('Line(7)={4,1};\n')
out.write('Line(8)={2,1};\n')
out.write('Line(9)={5,1};\n')

# left end, near vertical face
out.write('Line Loop(25)={-8,3,-4,9};\n')
out.write('Plane Surface(1)={25};\n')
# left end, far vertical face
out.write('Line Loop(26)={-9,6,-5,7};\n')
out.write('Plane Surface(2)={26};\n')
# left end, bottom face
out.write('Line Loop(27)={8,-7,2,-1};\n')
out.write('Plane Surface(3)={27};\n')

# right end of bar points
out.write('Point(8)={0,%f,%f};\n'%(L,H))
out.write('Point(9)={%f,%f,%f};\n'%(r45*math.cos(math.pi/4),L-
   r45*math.sin(math.pi/4),H))
out.write('Point(10)={0,%f,%f};\n'%(L-rs,H))
out.write('Point(11)={%f,%f,%f};\n'%(-r45*math.cos(math.pi/4),L-
   r45*math.sin(math.pi/4),H))
out.write('Point(12)={0,%f,%f};\n'%(L,H-rc))
out.write('Point(13)={%f,%f,%f};\n'%(ra*0.5*ct/al,L-ra*0.5*ct/al
   ,H-ra*0.5*(st**3+st+ct*ct*st)/al))
out.write('Point(14)={%f,%f,%f};\n'%(-ra*0.5*ct/al,L-ra*0.5*ct/
   al,H-ra*0.5*(st**3+st+ct*ct*st)/al))

# right end of bar lines
out.write('Ellipse(10)={9, 8, 10, 10};\n')
out.write('Ellipse(11)={11, 8, 10, 10};\n')
out.write('Ellipse(12)={9, 8, 13, 13};\n')
out.write('Ellipse(13)={12, 8, 13, 13};\n')
out.write('Ellipse(14)={11, 8, 14, 14};\n')
out.write('Ellipse(15)={12, 8, 14, 14};\n')
out.write('Line(16)={11,8};\n')
out.write('Line(17)={9,8};\n')
out.write('Line(18)={12,8};\n')

# right end, near vertical face
out.write('Line Loop(28)={18,-17,12,-13};\n')
out.write('Plane Surface(4)={28};\n')
# right end, far vertical face
out.write('Line Loop(29)={-18,15,-14,16};\n')
out.write('Plane Surface(5)={29};\n')
# right end, top face
out.write('Line Loop(30)={17,-16,11,-10};\n')
out.write('Plane Surface(6)={30};\n')

# crossing lines
out.write('Line(19)={3,12};\n')
out.write('Line(20)={2,13};\n')
out.write('Line(21)={6,9};\n')
out.write('Line(22)={5,10};\n')
out.write('Line(23)={7,11};\n')
out.write('Line(24)={4,14};\n')

# top 2 faces
out.write('Line Loop(31)={4,21,10,-22};\n')
out.write('Ruled Surface(7)={31};\n')
out.write('Line Loop(32)={-6,22,-11,-23};\n')
out.write('Ruled Surface(8)={32};\n')
# bottom 2 faces
out.write('Line Loop(33)={1,19,13,-20};\n')
out.write('Ruled Surface(9)={33};\n')
out.write('Line Loop(34)={-2,24,-15,-19};\n')
out.write('Ruled Surface(10)={34};\n')
# side faces
out.write('Line Loop(35)={-3,20,-12,-21};\n')
out.write('Ruled Surface(11)={35};\n')
out.write('Line Loop(36)={5,23,14,-24};\n')
out.write('Ruled Surface(12)={36};\n')

# the whole body
out.write('Surface Loop(13)={1,2,3,4,5,6,7,8,9,10,11,12};\n')
out.write('Volume(1)={13};')
```

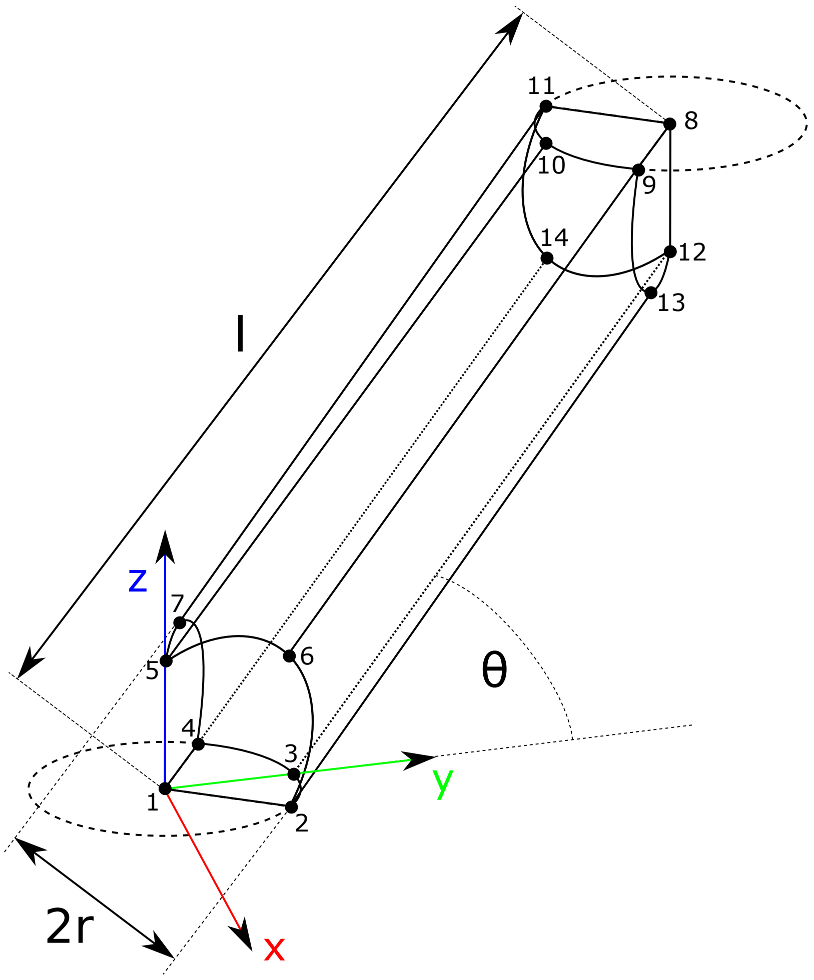


Figure A-1: Micro-truss leg BRep with axes and points labeled.

Bibliography

[1] W. Bangerth, R. Hartmann, and G. Kanschat. deal.ii–a general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software*, 33(4):24/1–24/27, 2007.

[2] R. Radovitzky, A. Seagraves, M. Tupek, and L. Noels. A scalable 3d fracture and fragmentation algorithm based on a hybrid, discontinuous galerkin, cohesive element method. *Computer Methods in Applied Mechanics and Engineering*, 200:326–344, 2011.

[3] T.J.R. Hughes. *The finite element method: Linear static and dynamic finite element analysis*. Dover Publications, Inc, New York, 2000.

[4] O. C. Zienkiewicz and R. L. Taylor . *The Finite Element method, 4th edn.* McGraw-Hill, New York, 1994.

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

[6] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3-4):237–254, 2006.

[7] A. Logg and G. N. Wells. DOLFIN: Automated finite element computing. *ACM Transactions on Mathematical Software*, 37(2):1–28, 2010.

[8] Junxian Liu, Paul Kelly, and Stuart Cox. Functional programming for finite element analysis, 1993.

[9] J. Besson and R. Foerch. Large scale object-oriented finite element code design. *Computer Methods in Applied Mechanics and Engineering*, 142:165–187, 1997.

[10] O. Pironneau, F. Hect, and A. Le Hyaric. Freefem++. http://www.freefem.org, 2009.

[11] Satish Balay, Jed Brown, , Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010.

[12] P.R. Amestoy, I.S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Applied Mechanics and Engineering*, 184(2-4):501 – 520, 2000.

[13] A. Gupta, S. Koric, and T. George. Sparse matrix factorization on massively parallel computers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 1:1–1:12, New York, NY, USA, 2009. ACM.

[14] A. Logg. Efficient representation of computational meshes. *International Journal of Computational Science*, 4(4):283–295, 2009.

[15] James Munkres. *Elements of Algebraic Topology.* Prentice Hall, 1984.

[16] Allen Hatcher. *Algebraic Topology.* Cambridge University Press, 2002.

[17] Top500.org. Top500 List - November 2010. http://www.top500.org/list/2010/11/100, November 2010.

[18] NVIDIA Corporation. NVIDIA CUDA Programming Guide, Version 2.0. http://developer.download.nvidia.com, November 2008.

[19] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[20] D. Luebke. CUDA: Scalable parallel programming for high-performance scientific computing. In *5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, 2008.*, pages 836 –838, May 2008.

[21] D.B. Kirk and W.H. Wen-mei. *Programming Massively Parallel Processors*. Morgan Kaufmann, 2010.

[22] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96:879 – 899, 2008.

[23] D. Goddeke, R. Strzodka, and S. Turek. Accelerating double precision fem simulations with gpus. *Proceedings of ASIM*, 2005.

[24] Dimitri Komatitsch, David Michea, and Gordon Erlebacher. Porting a high-order finite-element earthquake modeling application to nvidia graphics cards using cuda. *Journal of Parallel Distributed Computing*, 69:451–460, 2009.

[25] D. Goddeke, H. Wobker, R. Strzodka, J. Mohd-Yusof, P. McCormick, and S. Turek. Co-processor acceleration of an unmodified parallel solid mechanics code with feastgpu. *International Journal of Computational Science*, 4:254–269, 2009.

[26] Z.A. Taylor, M. Cheng, and S. Ourselin. High-speed nonlinear finite element analysis for surgical simulation using graphics processing units. *IEEE Transactions on Medical Imaging*, 27:650–663, 2008.

[27] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6:40–53, March 2008.

[28] A. Klockner, T. Warburton, J. Bridge, and J.S. Hesthaven. Nodal discontinuous galerkin methods on graphics processors. *Journal of Computational Physics*, 228:7863–7882, 2009.

[29] C. Cecka, A.J. Lew, and Darve E. Assembly of finite element methods on graphics processors. *International Journal for Numerical Methods in Engineering*, 85:640–669, 2011.

[30] NVIDIA Corporation. NVIDIA CUDA C Programming Best Practices Guide, Version 2.3. http://developer.download.nvidia.com, July 2009.

[31] R. Rabenseifner. Hybrid parallel programming on hpc platforms. In *Fifth European Workshop on OpenMP*, September 2003.

[32] D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for fem calculations on a gpu-enhanced cluster. *Parallel Computing*, 33:685–699, 2007.

[33] D. Komatitsch, G. Erlebacher, D. Goddeke, and D. Michea. High-order finite-element seismic wave propagation modeling with mpi on a large gpu cluster. *Journal of Computational Physics*, 2010.

[34] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In Association for Computing Machinery, editor, *Supercomputing*, San Diego, 1995.

[35] T. Pezeril, G. Saini, D. Veysset, S. Kooi, P. Fidkowski, R. Radovitzky, and K. A. Nelson. Direct visualization of laser-driven focusing shock waves (in press). *Physical Review Letters*, 2011.

[36] H.N.G. Wadley. Multifunctional periodic cellular metals. *Philisophical Transactions of the Royal Society A*, 364:31–68, 2006.

[37] L. Wang, M.C. Boyce, C-Y Wen, and E.L. Thomas. Plastic dissipation mechanisms in periodic microframe-structured polymers. *Advanced Functional Materials*, 19:1343–1350, 2009.

[38] M. Ruzzene and F. Scarpa. Directional and band-gap behavior of periodic auxetic lattices. *phys. stat. sol. (b)*, 242:665–680, 2005.

[39] L. Qiao, J. J. Rimoli, Y. Chen, C. A. Schuh, and R. Radovitzky. Nonlocal superelastic model of size-dependent hardening and dissipation in single crystal Cu-Al-Ni shape memory alloys. *Physical Review Letters*, 106(8):085504, 2011.

[40] Christophe Geuzaine and Jean-Franois Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.