

---

# **Local Time Stepping on High Performance Computing Architectures**

## **Mitigating CFL Bottlenecks for Large-Scale Wave Propagation**

Doctoral Dissertation submitted to the  
Faculty of Informatics of the Università della Svizzera Italiana  
in partial fulfillment of the requirements for the degree of  
Doctor of Philosophy

presented by  
**Max Rietmann**

under the supervision of  
Olaf Schenk

May 2015



---

## Dissertation Committee

<b>Michael Bader</b>	Technische Universität München, Munich, Germany
<b>Andreas Fichtner</b>	ETH Zürich, Zurich, Switzerland
<b>Marcus Grote</b>	Universität Basel, Basel, Switzerland
<b>Rolf Krause</b>	Università della Svizzera italiana, Lugano, Switzerland
<b>Igor Pivkin</b>	Università della Svizzera italiana, Lugano, Switzerland
<b>Olaf Schenk</b>	Università della Svizzera italiana, Lugano, Switzerland

Dissertation accepted on 12 May 2015

---

Research Advisor

**Olaf Schenk**

---

PhD Program Director

**Igor Pivkin**

---

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

---

Max Rietmann  
Lugano, 12 May 2015

*For my lovely wife*

The first principle is that you must not fool yourself and you are the easiest person to fool.

Richard P. Feynman

# Abstract

Modeling problems that require the simulation of hyperbolic PDEs (wave equations) on large heterogeneous domains have potentially many bottlenecks. We attack this problem through two techniques: the massively parallel capabilities of graphics processors (GPUs) and local time stepping (LTS) to mitigate any CFL bottlenecks on a multiscale mesh. Many modern supercomputing centers are installing GPUs due to their high performance, and extending existing seismic wave-propagation software to use GPUs is vitally important to give application scientists the highest possible performance. In addition to this architectural optimization, LTS schemes avoid performance losses in meshes with localized areas of refinement. Coupled with the GPU performance optimizations, the derivation and implementation of an Newmark LTS scheme enables next-generation performance for real-world applications. Included in this implementation is work addressing the load-balancing problem inherent to multi-level LTS schemes, enabling scalability to hundreds and thousands of CPUs and GPUs. These GPU, LTS, and scaling optimizations accelerate the performance of existing applications by a factor of 30 or more, and enable future modeling scenarios previously made unfeasible by the cost of standard explicit time-stepping schemes.



# Acknowledgements

The content of this thesis benefited greatly from the input and help of a number of people, including the coauthors of our published, accepted, and in-progress works. In particular, Piero Basini and Peter Messmer provided meshing and optimization help for our original GPU paper that motivated the GPU chapter. Daniel Peter's knowledge of SPECFEM3D was invaluable when implementing LTS. Bora Uçar's suggestion to use hypergraphs improved the work on load balancing our LTS implementation. The LTS work itself was greatly influenced by discussions and input from Marcus Grote. I thank my advisor Olaf Schenk for his academic and financial support, the University of Basel, the ETH Zurich, CSCS, and the USI Lugano for their support in all manners of my PhD. Finally, I thank my PhD committee, who agreed to take the time to provide useful feedback and questions regarding this thesis and the work leading up to it.



# Contents

<b>Contents</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Wave-Propagation Application: Seismology . . . . .	2
1.1.1 Earthquake simulation example . . . . .	3
1.1.2 Increased performance using graphics processors . . . . .	7
1.1.3 Mitigating time stepping bottlenecks . . . . .	7
1.2 Summary and Outline . . . . .	9
<b>2 Wave Propagation on Emerging Architectures</b>	<b>11</b>
2.1 GPU Background . . . . .	13
2.1.1 GPU vs. CPU architecture . . . . .	14
2.2 Roofline Model . . . . .	15
2.2.1 Matrix operation benchmarks . . . . .	17
2.2.2 CUDA programming model . . . . .	18
2.2.3 Related work . . . . .	19
2.3 SPECfEM3D GPU . . . . .	21
2.3.1 The spectral element method for GPUs . . . . .	22
2.3.2 SEM computational structure . . . . .	27
2.4 GPU performance optimizations . . . . .	29
2.5 Performance Evaluation . . . . .	38
2.5.1 Benchmarking GPU versus CPU simulations . . . . .	39
2.5.2 Weak scaling . . . . .	40
2.5.3 Strong scaling . . . . .	41
2.5.4 Large application test . . . . .	43
2.6 Roofline Model for SPECfEM3D . . . . .	45
2.6.1 Emerging architectures outlook . . . . .	47
2.7 Adjoint Tomography . . . . .	48
2.7.1 Adjoint methods overview . . . . .	48

2.7.2	Adjoint costs . . . . .	50
2.7.3	I/O optimizations . . . . .	51
2.7.4	Adjoint performance benchmarks . . . . .	54
2.8	Conclusion . . . . .	55
<b>3</b>	<b>Newmark Local Time Stepping</b>	<b>59</b>
3.1	Introduction . . . . .	60
3.2	Newmark Time Stepping for Wave Propagation . . . . .	61
3.2.1	Newmark time stepping . . . . .	62
3.2.2	Comparison with leapfrog . . . . .	65
3.2.3	Conservation of energy . . . . .	66
3.2.4	Fine and coarse element regions . . . . .	66
3.2.5	LTS-Newmark . . . . .	67
3.3	Single-Step Method . . . . .	69
3.3.1	LTS as modification to matrix <b>B</b> . . . . .	72
3.3.2	Equivalence to LTS-leapfrog . . . . .	73
3.4	Absorbing Boundaries . . . . .	73
3.4.1	Multiple refinement levels . . . . .	75
3.4.2	Single step equivalent for multiple levels . . . . .	78
3.5	LTS-Newmark for Continuous Elements . . . . .	80
3.5.1	Implementation detail . . . . .	83
3.6	Numerical Experiments . . . . .	84
3.6.1	Numerical convergence . . . . .	84
3.6.2	Stability and CFL invariance . . . . .	86
3.7	Implementation in Three Dimensions . . . . .	90
3.7.1	Implementation in SPECSEM3D . . . . .	90
3.7.2	Experiments in three dimensions . . . . .	91
3.7.3	LTS evaluation and validation . . . . .	91
3.7.4	LTS scaling . . . . .	92
3.8	Conclusions . . . . .	94
<b>4</b>	<b>Load-Balanced Local Time Stepping at Scale</b>	<b>95</b>
4.1	Introduction . . . . .	96
4.2	The Partitioning Problem . . . . .	96
4.2.1	LTS-Partition models: graphs and hypergraphs . . . . .	99
4.2.2	Partitioning algorithms for LTS . . . . .	103
4.3	Performance Experiments . . . . .	104
4.3.1	Application mesh benchmarks . . . . .	104
4.3.2	Partitioning results . . . . .	106

---

4.3.3	CPU and GPU performance results . . . . .	108
4.3.4	Cache performance . . . . .	111
4.3.5	Large example . . . . .	112
4.3.6	Application example: Tohoku . . . . .	113
4.3.7	Additional partitioning options . . . . .	115
4.4	Conclusion . . . . .	116
<b>5</b>	<b>Conclusions and Outlook</b>	<b>119</b>
5.1	Summary . . . . .	120
5.2	Revisited: Application Example . . . . .	122
5.3	Final Discussion . . . . .	125
	<b>Bibliography</b>	<b>127</b>



# **Chapter 1**

## **Introduction**

This thesis is presented as a work of *computational science*; a multidisciplinary effort bringing together computing and algorithmic advancements to greatly accelerate large-scale scientific applications. Although the work in this thesis is generally applicable, we are particularly focused on the efficient simulation of wave propagation at large computational scales — both in the problem size considered and the resources used. This is achieved in three parts:

1. high-performance implementations for newly developed graphics-processing computing architectures;
2. mitigating time-stepping bottlenecks via local time stepping (LTS);
3. load-balanced LTS for many-node supercomputing clusters.

Each of these points represents a separate contribution, each independently and dramatically increasing the performance of our newly developed additions to an existing software package. Critically, each component's performance can be combined to drastically improve performance over the original implementation.

The techniques developed and derived in this thesis can be generally applied; we have, however, specifically targeted computational seismology, which uses numerical computing to study seismic phenomena such as earthquakes. In particular, seismology is focused on the propagation and interaction of an earthquake's waves with the Earth's crust, mantle, and core. Computational seismology tries to simulate these waves using (quasi-)analytical and purely numerical methods. The large simulation domain (the entire Earth or portions of it) coupled with the large number of events simulated means that seismology has been pushing the boundaries of high-performance computing (HPC) for years. This thesis aims to both bring higher performance through the use of new classes of computing devices, but also to enable localized modeling that previously created drastic performance reductions due to time-stepping stability criteria. In order to motivate these enhancements, we detail an application example to highlight the computational costs and bottlenecks this thesis is trying to reduce or eliminate.

## 1.1 Wave-Propagation Application: Seismology

Like many scientific disciplines, seismology is driven forward by theoretical advances coupled to real data of increasing quality and volume. As seismologists collected more and better seismogram data (displacement, velocity, and acceleration recordings based on ground motion from earthquakes or other seismic sources), it became possible to characterize the propagation of seismic waves

through the Earth. In order to better understand the propagation of these waves, partial differential equations (PDEs) modeling acoustic and elastic domains were developed to describe the propagation of waves through the earth. In order to solve these models, various numerical techniques exist to simulate these waves.

From the perspective of a numerical modeler, the Earth can be represented by a surprisingly simple linear PDE-model, with a thin crust, a large elastic mantle, and a (mostly) reflecting core. 3D simulations run on such a model can agree, to an initial estimation, quite well with real recorded seismograms. However, by analyzing the fit between real and simulated data, we can make improvements to this simple earth model that potentially reveal interesting physical properties about the Earth's internal structure.

### 1.1.1 Earthquake simulation example

To make this discussion concrete, we present an example to compare real data recorded in Switzerland with a simulation using our seismic wave-propagation research package SPECFEM3D. To be more formally introduced in later chapters, SPECFEM3D [Peter et al., 2011] is a comprehensive software program to simulate the viscoelastic wave equation on hexahedral finite-element meshes on large-scale supercomputing systems. In other words, we can simulate earthquakes on a discrete domain to approximate and evaluate our spatial parameters (p- and s-wave velocities, density, etc) and source model (moment tensor, time, and location). With this in mind, we selected a real event and appropriate seismogram traces from recording stations to evaluate our ability to replicate the real data, with the setup seen in Fig. 1.1.

The two stations FUORN (east Switzerland) and SENIN (west Switzerland) recorded a magnitude ( $M_w$ ) 4.6 earthquake on September 08, 2005 at 11:27:22 AM, located in the Swiss-French Alps at a relatively shallow depth of 10 km. In order to simulate this earthquake in SPECFEM3D, we need a hexahedral mesh that covers the simulation domain with enough resolution to resolve the waves as they propagate from the source to the two receivers. The setup, with 5 km elements and the appropriate topography profile,<sup>1</sup> is shown in Fig. 1.2.

This mesh has about 350K elements and 23M degrees of freedom, due to the higher-order nature of the finite-element method used. With a stable time step  $\Delta t = 0.04$  s, we simulated 180 s in order to propagate the waves completely through the domain. Thus, we needed  $180/\Delta t = 4500$  steps, requiring 40 min of wall-clock time on a modern 8-core Intel CPU.

---

<sup>1</sup>Profile (SRTM) from <http://srtm.csi.cgiar.org>.

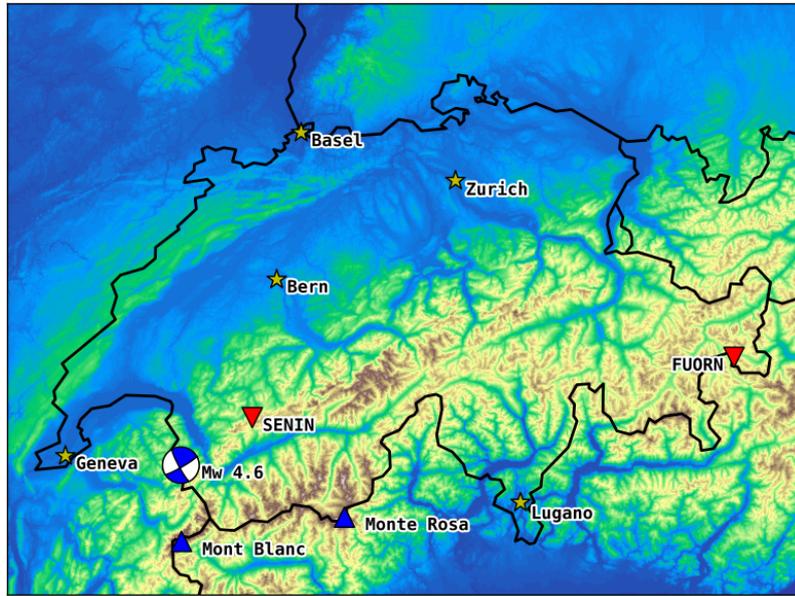


Figure 1.1. Earthquake simulation setup, with a Mw 4.6 earthquake in the Swiss-French Alps and two recording stations FUORN and SENIN.

For this example, we use a 1D or depth-only velocity profile [Diehl et al., 2009], which does not contain horizontal ( $x$ - $y$ ) variations. This depth-only velocity profile deviates only slightly from the well-known PREM model, which provides a global average profile and forms the basis of most velocity models.

Obtaining real data to compare with our simulation has also become much faster and easier than previously. Using the ObsPy seismology library [Beyreuther et al., 2010] to access the freely available IRIS database, we collected seismogram traces for the FUORN and SENIN stations for several minutes directly following the earthquake. This includes the respective instrument responses, allowing us to match our simulation to real, recorded data. After application of the instrument response, bandpass filtering between 60 and 10 s (1/60 and 1/10 Hz), and normalization we attained the seismogram comparisons for the two stations FUORN and SENIN for the  $z$ -component of the velocity seen in Fig. 1.3.

For a nonseismologist, the compared graphs may not look especially impressive, however, given that we are comparing a relatively simple model to data from a real earthquake recorded at a station in the mountains, the result is fairly promising. The energy that creates seismic waves, such as those seen here, propagates through the Earth along different ray paths. The first arrivals (the first small bump seen) travel through the Earth in an upwards parabolic trajectory (due to Snell's law). The slower surface velocity ensures that the larger

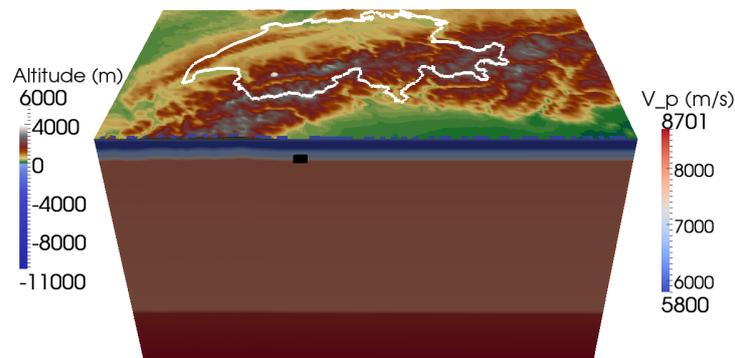


Figure 1.2. Hexahedral mesh domain with topography and our 1D (z-dir) velocity model covering our experimental setup in Switzerland. Mesh elements are approximately 5 km across. Note that p-wave velocity goes from 5800 m/s in the crust to 8700 m/s at a depth of 300 km.

surface-traveling waves arrive later (the stronger oscillations seen after the initial arrival). Thus, we see that the qualitative nature of the seismograms, which includes the arrival time of different wave components, is positive. Considering the FUORN graphs, the initial small bump, 40 s after the event, represents the direct wave, which travels through the medium and arrives at nearly the same time in both simulated and real data. The arrival of the larger peaks appearing later that represent surface waves is also very good. On the other hand, the general shape of the seismograms does leave room for improvement. Additionally, if we had raised the frequency cap on the bandpass to a higher cutoff, we would see further disagreement indicating that the smaller-scale variations are important and worth trying to improve for our horizontally homogeneous velocity model.

Although there are several techniques for improving the spatial velocity and density parameters (“the model”), we are particularly interested in a technique called *full waveform inversion* (FWI), which uses these simulations to construct linearized model updates, which are combined with nonlinear optimization techniques to iteratively improve the model by reducing the *misfit* (error) between real and simulated seismogram traces. A single earthquake event will not provide enough so-called “coverage” to improve the velocity model to a substantial degree. We will need to add significantly more earthquake sources to adequately resolve all of Switzerland. In Fig. 1.4, we map 168 earthquakes from the last 10–

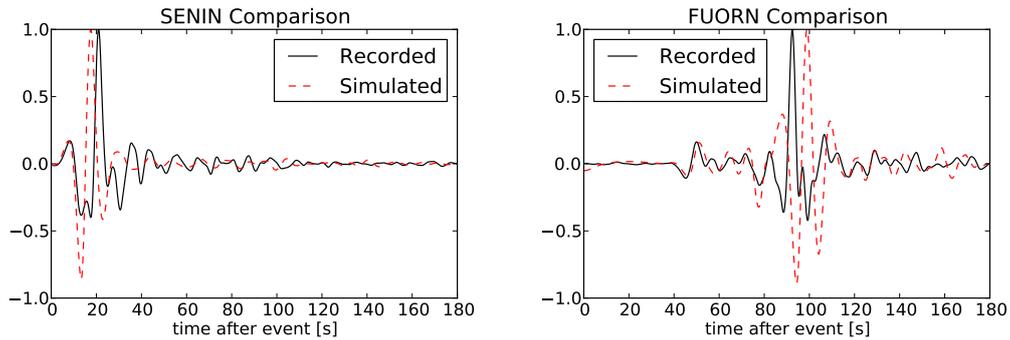


Figure 1.3. Comparison between real (recorded) and simulated (with SPECFEM3D) seismograms of z-component velocity for SENIN and FUORN. Filtered between 1/60 and 1/10 Hz and normalized.

15 yr, which could be used to do this for Switzerland. Each incremental model update requires several simulations *per earthquake*, and the iterative process can require many steps. Facing the cost of these many simulations, we are motivated to improve the performance of our simulation software to reduce the cost and pain associated with waiting for so many simulations.

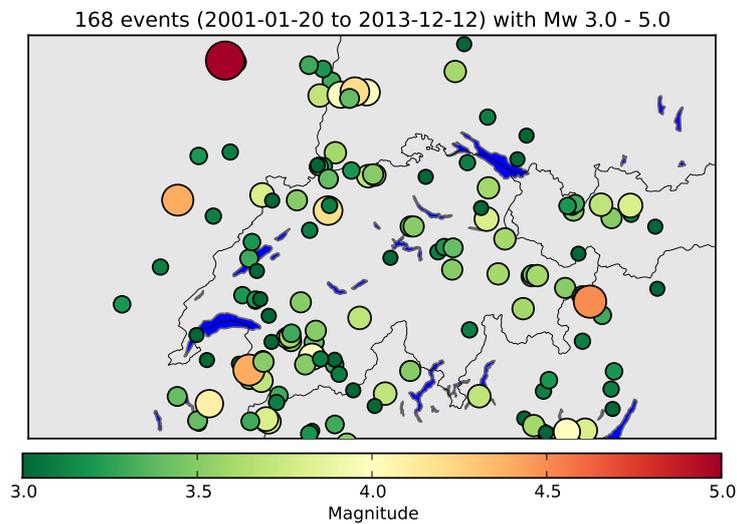


Figure 1.4. Earthquakes in Switzerland

### 1.1.2 Increased performance using graphics processors

At the beginning of this research, NVIDIA's newest compute-only graphics processors (GPUs) were just becoming popular enough that supercomputing centers were building small development clusters. It was hoped that these newly developed GPU "accelerators" would drastically improve performance for many fields in science, including computational seismology. Thus we have explored their use and detail the development of a GPU-capable software package for wave-propagation problems in Ch. 2. There we introduce extensions of the code to not only run these earthquake simulations, but also compute model updates to improve the match between real and simulated seismograms via adjoint tomography techniques, which should dramatically improve upon existing CPU performance.

To give a taste of these improvements, the simulation that required 40 min on a high-end 8-core CPU requires less than **6 min** of wall-clock time using a single NVIDIA Tesla K20c GPU. However, this performance comes at a high developer cost to port the existing CPU code to the GPU, which has a dramatically different programming model and set of critical optimizations in order to achieve the 6 min simulation time. We were able to match the wave-propagation computational structure to the threading model employed by GPUs, in addition to the cache and memory optimizations necessary to attain even adequate performance [Rietmann et al., 2012]. These performance improvements are, however, limited by the numerical techniques underlying the implementation. In particular, we are additionally interested in algorithmic advancements that can be coupled with the GPU speedup to yield double digit factors of improved performance.

### 1.1.3 Mitigating time stepping bottlenecks

Simply using the GPU improvements from Ch. 2, we can get a factor 5–7x shorter simulation times, drastically decreasing the time needed to update the model, or, conversely, increasing the coverage or detail we can achieve given an already fixed computational budget. However, as noted, the time-step size  $\Delta t$  factored directly into the cost (via the  $180/\Delta t = 4500$  required steps). The finite element mesh presented in this introduction is relatively homogeneous, despite the topography on the surface. In this Switzerland model, if we are interested in topography with additional detail, modeling of sediment layers, or any physical modeling that requires locally increased resolution, the impact on the time step  $\Delta t$  can drastically impact the simulation performance due to the CFL stability

criterion for standard explicit time-stepping schemes which specifies that

$$\Delta t \leq C_{\text{CFL}} h_{\text{min}}, \quad (1.1)$$

where  $h_{\text{min}}$  is proportional to the radius of the smallest element in the mesh and  $C_{\text{CFL}}$  is a constant that depends on the spatial discretization and time-stepping method. For this mesh in Switzerland, we additionally introduce a refinement around the source such that the smallest element is 32 times smaller than the largest, which can be seen in Fig. 1.5. This refinement results in a factor of 32x increase in simulation time due to the smaller time steps required to ensure stability when using a traditional time-stepping scheme. This translates to **2 hours 25 minutes** for the GPU (compared to 6 minutes), and more than **16 hours** for a single 8-core CPU (compared to 40 minutes). This drastic increase in simulation time is due to a very small fraction of the total elements — *only 3%* of the elements are 2x or more smaller than the bulk of elements in the mesh. This means that 97% of elements are taking a time step that is 32x smaller than needed.

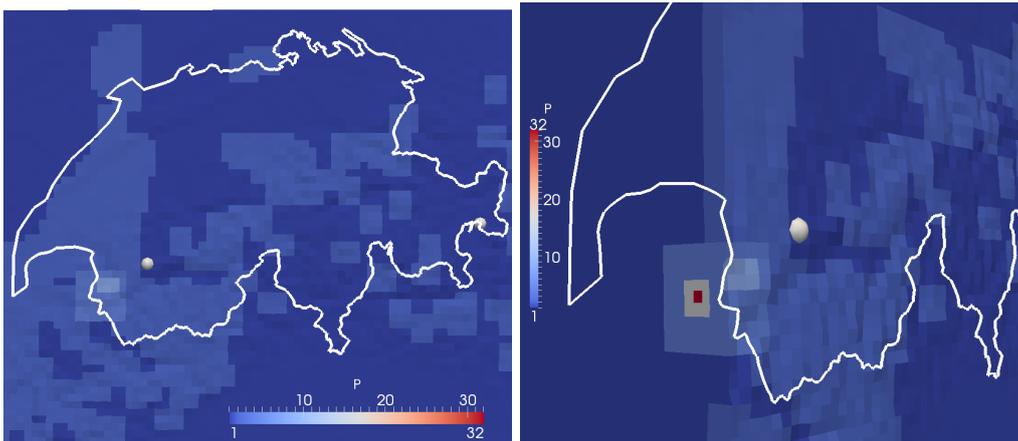


Figure 1.5. Refinement around source with smallest element  $h_{\text{min}} = h_{\text{max}}/32$ . The first layer of refinement can be seen close to the source location at the surface (left panel), and the zoom/cut (right panel) shows the smallest elements buried at the source location.

From a physical perspective, this simplistic refinement is difficult to motivate directly, but it does present a model for us to mitigate the drastic efficiency problems for a variety of practical applications. For example, in Ch. 4 we present an application where small elements arise when using a fault-rupture source model, which expands the traditional single-element earthquake source along a

fault, which must be honored by the finite element mesh, creating a localized region of small elements. Explicit time-stepping schemes such as the Newmark scheme (used in SPECFEM3D) are forced to take small time steps, which, as seen, dramatically increase simulation times. In Ch. 3 we derive an explicit LTS scheme that can take steps that are matched with the element size. This drastically reduces the amount of required computation, thus reducing the loss in performance due to the CFL stability criteria in (1.1).

Motivated by these applications in seismology, this thesis contributes not only a newly developed LTS-Newmark scheme for general use in wave propagation, but also an implementation on both CPUs and GPUs that can run on large-scale problems, with billions of degrees of freedom, within the well-known software package SPECFEM3D [Rietmann et al., 2014]. This LTS implementation has also been made parallel-ready [Rietmann et al., 2015], the contribution to be detailed in Ch. 4. This means we can run our LTS-Newmark scheme on meshes with millions of elements across hundreds of CPUs and GPUs, while maintaining parallel efficiency. As these contributions have been integrated into the SPECFEM3D package, they will be ready for use by real application seismologists, hopefully enabling applications that were previously too expensive due to the pure computational complexity, or due to CFL limitations from small elements.

## 1.2 Summary and Outline

We begin the thesis in Ch. 2 where we introduce the spectral finite element method, the seismology package SPECFEM3D, graphics processors (GPUs) for computing, and our GPU extensions to SPECFEM3D to take advantage of these emerging HPC devices. We present work that accelerates both earthquake simulations and tomography projects to improve the fit between data and simulation, as seen in Fig. 1.3. These improvements provide the first component of performance for a next-generation simulation package for general use in seismology.

Following the analysis and use of these emerging computing architectures, Ch. 3 develops the Newmark LTS algorithm for use in cases of localized element refinement, where time-stepping stability requirements can drastically reduce the application performance. We present both theoretical and experimental results to demonstrate the scheme's convergence and stability, but also specific algorithmic developments that are critical to an efficient implementation when using a continuous finite-element method. Using these techniques, the LTS-Newmark algorithm is efficiently implemented for both CPU and GPU architec-

tures in SPECFEM3D, providing 90%+ of single-threaded LTS efficiency.

Although the seismology example shown in this introduction is capable of running on a single GPU, it is critical that we are able to run with many additional CPUs and GPUs. Chapter 4 introduces the load-balancing problem that (multilevel) LTS creates and our solution that enables the use of more than 8000 CPUs on large problems with more than 1 billion degrees of freedom, a factor 100 larger than the example provided in this introduction.

Each chapter presents an independent advancement to the tools available to application scientists utilizing simulations of large-scale wave propagation. Significantly, these contributions will be generally available in the open-source SPECFEM3D code, bringing cutting edge simulation performance to computational seismologists, with only a runtime parameter setting. In the conclusion (Ch. 5) we will demonstrate the benefit of each chapter on the earthquake example provided in this introduction, demonstrating how coupled advances in algorithms and architectures deliver the next generation of performance needed by application scientists.

## **Chapter 2**

# **Wave Propagation on Emerging Architectures**

In 2003, Komatitsch et al. [2003] ran a global earthquake simulation with 14.6 billion degrees of freedom on the Japanese Earth Simulator, using 243 of 640 available nodes. At the time, the Earth Simulator was the number one super-computer (from TOP500), and the resulting paper from these experiments won the Supercomputing Gordon Bell prize. This simulation of the full 3D waveform on a global mesh was a great achievement and the SPECFEM3D software set the standard for efficient MPI-based wave propagation codes.

Since the initial development of SPECFEM3D, it has been split into two packages, one for purely global-scale propagation on a fixed, analytically defined mesh (SPECFEM3D GLOBE). The second, and the target for the work in this thesis, is SPECFEM3D Cartesian, which operates on a user-defined hexahedral mesh, which can be generated in a program such as Trelis (née CUBIT<sup>1</sup>). Both are well optimized to simulate the *forward model* very efficiently on CPUs and are now more limited by physical resolution limitations than by total cluster power or memory. In other words, increasing resolution of the underlying earth model (density, velocity, etc.) is an open research project and defines the current challenge in computational seismology.

As discussed in the introduction, improving the fit between real and synthetic (simulated) seismogram traces is an important application domain in seismology. This process of imaging the Earth using data from earthquakes and other sources, called *tomography*, is an *inverse* procedure that tries to match simulated (synthetic) data to real earthquake seismogram recordings. Following the adjoint tomography procedure [Tromp et al., 2004; Peter et al., 2007], many groups are able to utilize existing forward codes, make relatively small changes, and run the many required simulations to calculate a gradient that describes how to iteratively update the earth's velocity model. Using gradient-only nonlinear optimization techniques such as nonlinear conjugate gradient, they are able to update the 3-D seismic velocities, progressively reducing the misfit between the simulations and their real seismogram counterparts measured from actual events.

To do adjoint tomography using the SPECFEM3D package, a database of earthquakes is assembled, corresponding to seismograms recorded at stations within the region of interest. For each earthquake in the database, a forward simulation is compared with actual seismogram data to produce the adjoint source at each recording station. For the second step, both the forward and adjoint fields are simulated, which are compared in order to produce the Fréchet deriva-

---

<sup>1</sup>Trelis (originally CUBIT) is a commercially available software package that specializes in mesh generation using hexahedral elements and is currently available through CSimSoft (<http://www.csimsoft.com>).

tives, or gradient used in the optimization scheme. Each earthquake-source gradient is summed to provide the best spatial coverage possible, which is typically given as input to a nonlinear optimization technique such as conjugate gradient or BFGS [Wright and Nocedal, 1999] in order to update the model. Given an appropriate step length, the model update will reduce the misfit between our real data and the simulation. This iterative process continues until the stopping criterion is reached, which can be defined by a sufficiently small change in misfit.

Previous regional and global tomographic experiments have shown that this method can converge after 10–30 iterations [Fichtner et al., 2009; Tape et al., 2009; Zhu et al., 2012]. We can thus estimate that the final model will require approximately

$$3 \times (\text{number of earthquakes}) \times (20 \text{ steps})$$

equivalent forward simulations. For a database of 150 earthquakes, we will have 9000 simulations to perform. Depending on the scale of meshes in question, a full inversion using this adjoint tomography procedure requires at least 2 million CPU hours for 150 earthquakes on a smaller regional mesh, and > 700 million CPU hours using a large database of high quality earthquake data of the last ten years on the global scale. Beyond the purely computational challenges associated with these imaging methods, they are inherently ill-posed inverse problems, making them sensitive to details of the methods themselves. It is very common to try and compare several combinations of data filters, preconditioners, update schemes, and regularization terms when updating the model, requiring further sets of simulations, compounding an already difficult problem.

Many levels in the algorithm and software stack of this inverse problem are active areas of research. In this chapter we specifically present the acceleration of the forward and adjoint simulations using graphics processors (GPUs) to increase SPEC-FEM3D's performance, thus reducing the time-to-solution for specific forward and inverse seismology problems.

## 2.1 GPU Background

Before looking at our SPEC-FEM3D implementation, it will be useful to look at a brief history of CUDA, GPU computing, the gains that others have achieved, and the performance we expect.

### 2.1.1 GPU vs. CPU architecture

The GPU implementation of SPECfem3D is done using NVIDIA's CUDA programming model, which requires that the programmer maintain a mental model of the GPU as a computing device in order to get reasonable performance from the device. Therefore we will briefly introduce NVIDIA's GPU architecture and their CUDA programming model in order to better understand our SPECfem3D GPU implementation.

In order to see how a GPU can improve performance for a scientific workflow, it is best to consider its origins. The name, *graphics processor*, reveals a lot about what makes a GPU special and different from a traditional multicore CPU. GPUs are purpose built graphics rendering computing devices. Fortunately for us, NVIDIA and others introduced the ability to add programmable logic to their graphics pipelines, initially used to create more spectacular effects in video games. Some of the initial experiments in GPU scientific computing were done by framing a scientific problem in terms of this programmable rendering pipeline.

NVIDIA saw a potential market opening and released CUDA 1.0 concurrently with new GPUs that could be programmed using this new language and associated libraries. They also released the initial "Tesla" model GPU, built to withstand the nonstop workloads and additional heat requirements typical of a scientific cluster. The CUDA programming model made both scientific computing on GPUs much easier than the rendering pipeline "hack," it also increased the available performance. Since the initial release, NVIDIA has released several new GPU micro-architectures, and many versions of CUDA compilers, libraries, debugging, and profiling tools. Their "Kepler" generation GPUs power the world's #2 supercomputer and soon, CUDA capable GPUs will be running in smartphones, putting hundreds of GFLOP/s of performance literally in the palm of your hand.

Although superficially similar, a GPU and a CPU are built for very different workloads and performance characteristics. A modern 8-core CPU from Intel contains around 2 billion transistors, where a modern Kepler GPU from NVIDIA has closer to 7 billion. These 2 billion transistors on a CPU make up each core, caching logic, and the multiple cache layers. Each core contains complex instruction logic to decode, rearrange, and predict the program flow, and also integer and vector floating point pipelines. They allocate significant resources to advanced features such as out-of-order execution, which can rearrange the program stream to avoid memory stalls, and sophisticated branch prediction, which tries to estimate branching logic (e.g., 'if' statements) to avoid pipeline stalls by

preloading the predicted branch's instruction stream. All of these components work together to provide the best performance across all types of workloads — numerical algorithms or word processing.

A GPU, by comparison, generally does not allocate space for any of these sophisticated control mechanisms such as instruction reordering and branch prediction. This simplicity leaves room for additional registers and ALUs, potentially giving GPUs a huge advantage on numerical tasks where the execution stream is predictable and data parallel. In practice, as noted by Volkov and Demmel [2008], we should view the GPU as a simple CPU with very long vector units, much like the vector machines of the 1980s and 1990s. Many numerical and scientific applications can be framed using this parallelism model which is reflected by the impressive performance one can see at any conference or expo where GPU-accelerated scientific applications are exhibited. In order to set our expectations about what GPUs can bring SPECfem3d, we turn to the roofline performance model [Williams et al., 2009] and linear algebra benchmark comparisons to model the performance of GPUs relative to CPUs.

## 2.2 Roofline Model

Because scientific applications usually run in a supercomputing setting, a comparison between the original CPU version, and a GPU version should always be conducted on a node-to-node basis. That is, a CPU node and a GPU node of the same generation, which usually consists of two sockets of 6–8 cores each, or a single socket and a single GPU. It is also useful to get an upper bound on the speedup an application will see, which we can estimate using CPU and GPU BLAS libraries. Large matrix-matrix multiplication is a near ideal use-case for GPUs, and the performance of a single Tesla GPU is impressive.

The finite-element method implemented by SPECfem3d can be modeled as a sparse-matrix vector product  $\mathbf{K}\mathbf{u}$ , with a series of pointwise vector operations to complete the time step. By benchmarking synthetic examples of matrix-matrix, matrix-vector, vector-vector multiplication, we can both model how much speedup we can expect via roofline modeling and, once we have SPECfem3d GPU benchmarks, determine how close we are to the peak achievable performance for our particular situation.

The data sheet for the newest NVIDIA K20X accelerators lists peak performance for single precision at near 4 TFLOP/s. By this metric alone, this single device would have been the #1 supercomputer in November 1999. Using a highly tuned version of CUBLAS, SGEMM (single-precision matrix-matrix multi-

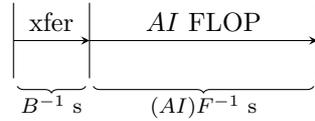
ply) benchmarks show that the device is able to achieve only 73% of this peak, indicating that real-world applications will likely achieve even less of the theoretical performance. Also to note is the fact that the performance of many algorithms (and their implementations) are limited by memory bandwidth, and thus we would like to model the performance of a given algorithm using the two metrics of peak performance and memory bandwidth.

We turn to the *roofline* model introduced by Williams et al. [2009]. This model requires us to determine the arithmetic intensity<sup>2</sup> ( $AI$ ), which relates the FLOP done per byte transferred,

$$AI = \frac{\text{FLOP}}{\text{bytes transferred}}.$$

The bytes transferred takes caching into account — a value that is found in cache that avoids a memory transfer is not included in the count of bytes transferred. This is potentially difficult to count correctly on a CPU where cache usage is implicit, but is much easier to calculate on the GPU because the use of cache (shared memory) is done mostly by hand.

We define  $F$  as the peak performance (FLOP/s), and  $B$  as the device bandwidth (bytes/s). Let us consider the time to compute an operation on a single byte,



First the byte must be transferred from main memory, followed by  $AI$  FLOP.  $AI$  defines the amount of work that must be done on this byte, and given a simple definition for performance, we can quickly write a formula for performance based on our device given  $B$  and  $F$  and the algorithm's  $AI$

$$\text{Performance } (P) = \frac{\text{FLOPs}}{\text{seconds}} = \frac{AI}{B^{-1} + F^{-1}(AI) s} \quad (2.1)$$

Intuitively, for small  $AI$ , memory bandwidth  $B$  is the limiting factor, where for large  $AI$ , the peak floating point performance limits total performance. We thus simplify (2.1), where the equation for performance is approximately linear for low  $AI$  and approximately constant for large  $AI$ :

$$P(AI) \approx \begin{cases} B(AI), & 0 < AI < F/B \\ F, & AI > F/B. \end{cases} \quad (2.2)$$

<sup>2</sup>The original paper uses the term *operational intensity*, which we replace with  $AI$ , however with the same definition.

The two approximations meet at  $AI = F/B$ , which is the point where peak floating point performance starts limiting total performance more heavily than peak memory bandwidth.

### 2.2.1 Matrix operation benchmarks

The roofline model allows the user to evaluate the possible peak performance of their application or algorithm. This is especially important when considering a GPU version of a code, which might bring significantly higher performance, but at additional development and maintenance cost — for a working scientist, the trade-off between performance and productivity is always present. The rise of easy-to-use languages and libraries such as Matlab, R, NumPy/SciPy, and Julia demonstrate the productivity part of this spectrum, as compared to classical Fortran and C development. However the typical scale of wave propagation and related problems requires a high-performance implementation, with at least the core developed in Fortran or C/C++.

In order to build a roofline model to compare CPU and GPU performance, we construct the model using measured performance numbers from benchmarking BLAS1, BLAS2, and BLAS3 vector and matrix operations. In Fig. 2.1 we compare an 8-core Intel E5-2670 processor and an NVIDIA K20x GPU, and use Intel’s highly tuned MKL BLAS library, and NVIDIA’s CUBLAS library, to implement single-precision dot product ( $x^T y$ ) (BLAS1), matrix-vector multiplication ( $Ax$ ) (BLAS2), and matrix-matrix multiplication ( $AB$ ) (BLAS3), such that matrix-matrix multiplication becomes our “peak” floating point performance. Memory bandwidth was measured using an adapted version of the STREAM<sup>3</sup> benchmarking code and was around 150 GB/s (with ECC on) for the K20x and around 50 GB/s for the CPU.

As can be seen and expected, the speedup is close to 3x for the lower arithmetic intensity BLAS1 and BLAS2 operations follow the bandwidth curves. For large matrix-matrix multiplication, we expect and achieve greater final speedup of 6.7x, which fits the ratio between NVIDIA’s and Intel’s published theoretical peak performance. Thus, if our algorithm can be structured to take advantage of cache and achieve a relatively high arithmetic intensity, we can achieve a GPU speedup between 3x and 7x (relative to 8 cores). However, as most supercomputing CPU nodes have *two* 8-core CPUs (compared to only a single GPU) a more realistic speedup<sup>4</sup> is 2x–4x to provide a more realistic evaluation. Of course this

<sup>3</sup><https://www.cs.virginia.edu/stream>

<sup>4</sup>2x–4x instead of 1.5x–3.5x to account for CPU scaling inefficiencies.

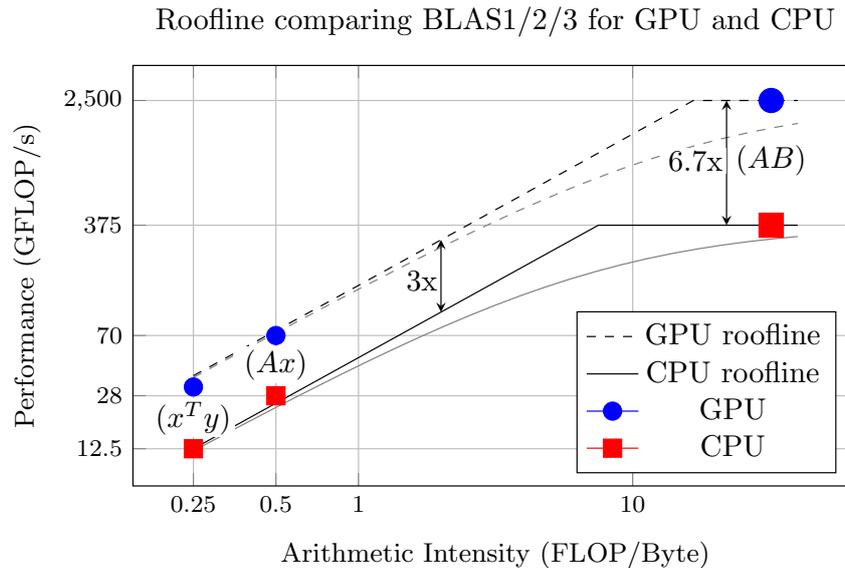


Figure 2.1. Exact roofline model (2.1) and first-order approximation (2.2) comparison between BLAS1, BLAS2, and BLAS3 operations on a single K20x GPU vs. an 8-core Intel CPU using the highly tuned linear algebra libraries CUBLAS and Intel MKL.

assumes that both the GPU and CPU versions run a similarly well-optimized version of the same algorithm.

### 2.2.2 CUDA programming model

Many codes looking to add performance via GPUs are written in Fortran or C/C++ and utilize MPI for both intranode and internode parallelization. In order to push into the area of multicore and many-core architectures, some codes are exploring shared memory threading for intranode parallelism in order to reduce the communications load resulting from too many MPI processes. On the CPU side, we see OpenMP as the dominant threading backend, for both multicore CPUs and the new many-core Xeon Phi (MIC) from Intel.

The CUDA programming model exposes the GPU as a shared memory massively-threaded computing device, and in contrast to OpenMP, the CUDA C language is a superset of the C language as opposed to a set of compiler directives. CUDA allows the programmer to specify functions that should execute on the GPU called *kernels*. CUDA includes additional syntax, for both kernel specification and launch on the GPU, using any number of desired threads.

In a typical CPU shared-memory data parallel application, a threading model like OpenMP is used to parallelize a loop, where the iterations are independent. Output variables and arrays where reduction operations can occur require special care using atomic operations or locking mechanisms. To achieve the best performance, most programmers maintain a mental model of the loop iterations, and how the work is being split across CPU threads. We see a very small abbreviated example implementation using OpenMP (left) and CUDA (right) of the operation  $\vec{z} \leftarrow a\vec{x} + \vec{y}$  in Fig. 2.2.

```

// OpenMP parallelization
void saxpy_cpu(a,x,y,z)
#pragma omp parallel for
for(int i=0; i<N; i++)
    z[i] = a*x[i] + y[i];

// CUDA Kernel
__global__ void saxpy_kernel(a,x,y,z)
int id = threadIdx.x
        + blockIdx.x*blockDim.x;
z[id] = a*x[id] + y[id];

```

Figure 2.2. Abbreviated C-code comparing OpenMP and CUDA implementations of  $\vec{z} \leftarrow a\vec{x} + \vec{y}$ .

The CUDA GPU programming model, when framed in terms of threading a looping construct, is remarkably similar. The major difference is that CUDA is *always* parallel and the loop keywords (e.g., `for(;;)`) are never actually written. Generally the body of the loop is written as a function of the loop index (in this case `id`), where one thread is launched per loop index and identified using the thread unique structures `threadIdx` and `blockIdx`. These threads, in contrast to CPU threads, are very lightweight and work in teams called blocks and are scheduled in groups called “warps.” These warps work together to coalesce memory transfers and cache access. Ensuring that variables are organized to take advantage of the warp collective access patterns is critical to performant GPU code. Additionally, as in OpenMP or any shared-memory threading model, CUDA kernels must also take care to avoid race conditions, an important performance consideration we will see in an upcoming section.

### 2.2.3 Related work

Within a few years of the release of CUDA, many groups began to experiment with GPUs for scientific and numerical work, many especially interested in the simulation of PDEs, especially for hyperbolic problems such as wave propagation. Klöckner et al. [2009] demonstrated a newly developed higher-order discontinuous Galerkin (DG) GPU code simulating Maxwell’s equations. Comparing a single CPU against a single consumer GPU, they managed a speedup of

65x. However, an experiment conducted using supercomputing resources (of that time) would have pitted a single GPU (of similar performance) against two quad-core CPUs. Assuming perfect CPU scaling, we would have seen a speedup closer to 8x, which suggests that this implementation of higher-order (discontinuous) finite elements yields more than the expected speedup and that the CPU version was less than optimal.

More recently, the HPC world has seen larger-scale GPU cluster installations and corresponding simulations, such as work presented by Shimokawabe et al. [2011]. Running on Japan’s TSUBAME 2.0 cluster with 4,000 GPUs (and 16,000 CPU cores), they ran a “dendritic solidification” simulation that achieved just over 1 PFLOP/s of single-precision performance using the full cluster. Although different than the hyperbolic PDE models we are interested in solving, the PDE model consists of separated spatial and temporal differential operators, allowing finite differencing in space and an explicit time-stepping scheme in time to be used. A traditional parallelization method is used, where the physical domain (a finite difference grid) is partitioned across GPUs. However, in an attempt to utilize the CPUs on each node, they compute these halo-region elements using the CPUs in parallel with the “inner” grid elements on the GPU. The MPI messages are sent by the CPUs as it completes the top, bottom, back, front, left, and right faces. Thus, this approach can utilize the full compute power of the system, CPUs + GPUs, where many GPU simulations (including SPEC-FEM3D) leave the CPUs essentially idle, used only to control the GPU simulations and manage I/O for MPI and disk operations.

Finally, we consider the SEISSOL software package [Dumbser and Käser, 2006]. Providing a similar set of features to the SPEC-FEM3D package, it simulates the elastic wave equation on unstructured finite-element meshes. It additionally provides a form of local time stepping [Dumbser et al., 2007] (developed for SPEC-FEM3D in Ch. 3). In contrast to the *continuous* finite-element spatial discretization employed by SPEC-FEM3D, SEISSOL utilizes a DG finite-element method, which has several advantages and disadvantages. Because the finite-element polynomials are discontinuous at each element boundary, the mass matrix  $\mathbf{M}$  is block diagonal, where each element’s block in the matrix is fully uncoupled from the others. Thus, it is no longer necessary to employ the use of Gauss–Lobatto–Legendre (GLL) collocation points and Gaussian quadrature [Komatitsch et al., 2003] to achieve the efficient computation of  $\mathbf{M}^{-1}$  (trivial in SPEC-FEM3D because  $\mathbf{M}$  is diagonal). This allows the use of both hexahedral and *tetrahedral* elements, where tetrahedra are generally far easier to use from a mesh-generation perspective. This additional flexibility can be a big advantage, depending on the particular use. However, the duplicated degrees of freedom

at each element boundary come with a computational cost, and SEISSOL was known to be less efficient than SPECFEM3D, although it is difficult to find an exacting comparison between the two.

Recently, Heinecke et al. [2014] detailed the optimization and extension of SEISSOL to work using Intel’s XEON Phi (MIC) accelerator modules, which are Intel’s response to GPU computing and have many similarities from an architectural perspective, including a large number (60+) of relatively simple processors with wide vector units and high bandwidth GDDR5 memory. Experiments run on the Tianhe-2 supercomputer (world’s fastest as of November 2014 and largest Intel MIC cluster installation) with three Intel MIC accelerators in each of the 8192 nodes, was able to achieve up to 8.6 PFLOP/s of double-precision performance in weak-scaling benchmarks. SEISSOL follows a similar parallel structure to SPECFEM3D (see Sec. 2.4), where partition boundaries are computed first in order to allow the MPI communications to run concurrently with the “inner” nonboundary element updates.

## 2.3 SPECFEM3D GPU

The initial excitement about the potential performance of GPUs prompted Komatitsch et al. [2010] to port a limited version of the SPECFEM3D forward solver to GPUs. They showed a speedup of 13x using a single NVIDIA Tesla S1070 against a 4-core Intel “Nehalem” CPU — far less than the incredible speedups in some fields, but a promising result nonetheless. This limited code, however, was not meant to be used in production, and would require significant effort to extend for use in seismic imaging.

In order to reevaluate the code on the newest GPU clusters and extend the functionality to include the additional routines necessary for adjoint tomography, we decided to add GPU functionality to the SPECFEM3D 2.0 “Sesame” [Peter et al., 2011] software package (via a `GPU_MODE` logical). We were able to reintegrate the stiffness matrix and time-stepping GPU kernels, but the GPU initialization and extensions for adjoint tomography were additionally ported to CUDA using the CPU version as a reference. Although this added additional complexity to the original code, using the structure of the CPU version to stage the GPU computations makes both testing and maintenance much simpler.

We choose to use CUDA as the GPU programming language and environment, given its maturity, stability, and ubiquity at the time of development.<sup>5</sup> The

---

<sup>5</sup>Mixing CUDA-C and Fortran did, however, create a lot of off-by-one bugs due to their array indexing differences.

rise of semiautomatic schemes using accelerator directives, such as OpenACC, seems to be an excellent compromise, but were seen as too immature at the time of development. Additionally, they can make use of NVIDIA's additional profiling and developer tools more difficult. It is also unlikely that one could port the entire SPECFEM3D code to GPUs via OpenACC and expect excellent performance; however, a hybrid approach may provide the shortest time-to-solution in terms of development costs. By initially annotating the entire time-stepping loop with OpenACC directives, the initial GPU version can be built and tested with relative confidence. By profiling, we can determine the most complex routines which can be rewritten by hand using CUDA, saving a lot of development effort for the many simpler code paths, such as time stepping, absorbing boundaries, and simple source mechanisms, which are simple enough for an OpenACC compiler.

We might also consider semiautomated CUDA compilers such as PyCUDA which was developed by Klöckner et al. [2012]. This unique CUDA framework allows the generation of (optimized) CUDA kernels from within a simplified Python DSL. The use of Python in science via NumPy and SciPy has grown rapidly, and provides compelling competition for native Fortran or C++ CUDA development due to Python's ease of use and extensive and high-quality scientific and numerical library ecosystem. PyCUDA uses C-code generation techniques to provide high-performance CUDA kernels; however, it is hoped that NVIDIA's release of an LLVM<sup>6</sup> → PTX<sup>7</sup> library will allow many more tools similar to PyCUDA that make GPU programming drastically easier. Holk et al. [2013] already presented a proof of this concept via the Rust<sup>8</sup> language and compiler, which uses LLVM as its intermediate representation (IR) and optimizing backend. With only a few modifications and the addition of several keywords, they were able to write CUDA kernels in Rust that were compiled into native CUDA kernels with performance comparable to writing them in CUDA C.

### 2.3.1 The spectral element method for GPUs

Having introduced the CUDA programming model, we can introduce how we implemented our particular choice of spatial discretization for applications in seismology. SPECFEM3D implements the spectral element method (SEM), which is a carefully constructed finite-element method that provides a flexible and ef-

---

<sup>6</sup>LLVM is a general-purpose intermediate representation and backend for compiler authors ([www.llvm.org](http://www.llvm.org))

<sup>7</sup>PTX is a GPU-specific intermediate representation for NVIDIA GPUs

<sup>8</sup>A new systems language with a focus on memory safety, concurrency, and performance (<http://www.rust-lang.org>).

ficient spatial discretization for wave-propagation modeling. More specifically, we are interested in discretizing the elastic wave equation on some 3D domain  $\mathbf{x} \in \Omega$ ,

$$\rho(\mathbf{x}) \frac{\partial^2 \mathbf{u}}{\partial t^2} - \nabla \cdot \mathbf{T}(\mathbf{x}, t) = f(\mathbf{x}_s, t), \quad (2.3)$$

$$\mathbf{T}(\mathbf{x}, t) = \mathbf{C}(\mathbf{x}) : \nabla \mathbf{u}(\mathbf{x}, t), \quad (2.4)$$

where  $\rho(\mathbf{x})$  is the material density,  $\mathbf{C}$  is the forth-order elasticity tensor with 21 independent parameters in the fully anisotropic case and  $f(\mathbf{x}_s, t)$  represents the forcing caused by e.g., an earthquake, commonly modeled as a point source at  $\mathbf{x}_s$ . Additionally, for well-posedness, this equation requires appropriate initial and boundary conditions. In order to motivate the structure of the code, and associated computational costs, we present an abbreviated outline of the SEM applied to (2.3), which can be seen in far greater detail from the original creators of SPEC-FEM3D, Komatitsch and Tromp [2002]; Tromp et al. [2008], and from Canuto et al. [2006].

Following a standard Galerkin finite-element procedure we multiply this equation by a test function  $\mathbf{w}$  and integrate by parts over the domain  $\Omega$  to obtain the weak formulation

$$\int_{\Omega} \mathbf{w} \cdot \rho(\mathbf{x}) \frac{\partial^2 \mathbf{u}}{\partial t^2} = \int_{\partial\Omega} \hat{\mathbf{n}} \cdot \mathbf{T} \cdot \mathbf{w} dS - \int_{\Omega} \nabla \mathbf{w} : \mathbf{T} d\Omega + \int_{\Omega} \mathbf{w} \cdot f(\mathbf{x}_s, t) d\Omega. \quad (2.5)$$

We note that the first and third terms of this expression will yield the so-called *mass* and *stiffness* matrices. The second term, an integral over the domain's boundary surfaces, is typically modeled as zero on the free surface, and is reformulated on the artificial domain boundaries as an (approximate) absorbing boundary condition. The final term including the source, is generally localized to a single element, although we will consider an application with a fault source in Sec. 4.3.6.

We begin the discretization process by representing the 3D domain  $\Omega$  with a set of nonoverlapping hexahedral elements  $\Omega_k$  to make up our discrete domain such that

$$\bigcup_{k=1}^K \Omega_k = \Omega_h.$$

We further restrict the space of  $\mathbf{w}$  to a finite set of polynomials  $\phi(x, y, z)$  each with compact support in each element  $\Omega_k$ . To simplify defining the method, consider a reference element defined by the variables  $-1 \leq \xi, \eta, \zeta \leq 1$ . For the

SEM, we choose our polynomials as the set of Lagrange polynomials defined by

$$l_j(\xi) = \prod_{\substack{0 \leq m \leq N \\ m \neq j}} \frac{\xi - \xi_m}{\xi_j - \xi_m} = \frac{(\xi - \xi_0)}{(\xi_j - \xi_0)} \cdots \frac{(\xi - \xi_{j-1})}{(\xi_j - \xi_{j-1})} \frac{(\xi - \xi_{j+1})}{(\xi_j - \xi_{j+1})} \cdots \frac{(\xi - \xi_k)}{(\xi_j - \xi_k)}$$

for  $0 \leq j \leq N$ , where we note that  $l_j(\xi_j) = 1$  and  $l_j(\xi_{i \neq j}) = 0$ . If we consider a function  $f(\mathbf{x})$ , we represent it using the following approximation using triple products of these Lagrange polynomials

$$f(\mathbf{x}(\xi, \eta, \zeta)) \approx \sum_{i,j,k=0}^{N_i, N_j, N_k} f_{i,j,k} l_i(\xi) l_j(\eta) l_k(\zeta).$$

where  $f_{i,j,k} = f(\mathbf{x}(\xi_i, \eta_j, \zeta_k))$ . Critical to the SEM, we choose our set of  $(\xi_i, \eta_j, \zeta_k)$  to be the well-known GLL collocation points, which when combined with these Lagrange polynomials and the GLL quadrature rule, simplify the evaluation of the integrals in the weak formulation and critically yield a *diagonal* mass matrix allowing for explicit time-stepping schemes.

In order to solve the weak form (2.5) using our reduced set of test functions, we utilize the GLL quadrature such that the following integral approximation on an element  $\Omega_e$  retains the convergence properties of a standard finite-element method

$$\int_{\Omega_e} f(\mathbf{x}) d\Omega = \int_{\Omega_{\text{ref}}} f(\mathbf{x}(\xi, \eta, \zeta)) J(\xi, \eta, \zeta) d\Omega_{\text{ref}} \quad (2.6)$$

$$\approx \sum_{i,j,k=0}^{N_i, N_j, N_k} \omega_i \omega_j \omega_k f_{i,j,k} J_{i,j,k} \quad (2.7)$$

where  $\omega_{i,j,k}$  represent the GLL quadrature rules and  $J(\xi, \eta, \zeta)$  represents the integral mapping between the reference element  $\Omega_{\text{ref}}$  and the original element  $\Omega_e$ . With the integral rule defined, we can apply this to our original weak form. The first term in the weak form (2.5), after integration, becomes the so-called *mass matrix*. We can rewrite this term with respect to a single element  $\Omega_e$  and the reference element  $\Omega_{\text{ref}}$ , and with the application of (2.6), we get the final spatially discrete form

$$\begin{aligned} \int_{\Omega_e} \mathbf{w} \cdot \rho(\mathbf{x}) \frac{\partial^2 \mathbf{u}}{\partial t^2} d\Omega &= \int_{\Omega_{\text{ref}}} \rho(\mathbf{x}(\xi)) \mathbf{w}(\mathbf{x}(\xi)) \cdot \frac{\partial^2 \mathbf{u}(\mathbf{x}(\xi))}{\partial t^2} J(\xi) d\Omega_{\text{ref}} \\ &\approx \sum_{i,j,k=0}^{N_i, N_j, N_k} \omega_i \omega_j \omega_k J_{i,j,k} \rho_{i,j,k} \sum_{m=1}^3 \mathbf{w}_{i,j,k}^{(m)} \frac{\partial^2 \mathbf{u}_{i,j,k}^{(m)}}{\partial t^2} \end{aligned} \quad (2.8)$$

Following the Galerkin finite-element method, we choose  $\mathbf{w}_{i,j,k} = \ell_{i,j,k}(\xi)$ , each of which is nonzero only at a single GLL collocation point, that is,  $w^{(m)}$  can be chosen to be independently zero, creating independent expressions for each component of  $\mathbf{u}$ . Critically, this resulting set of expressions is assembled (in the finite-element sense) into the diagonal matrix  $\mathbf{M}$ , allowing the use of explicit time-stepping schemes for time-discretization.

Continuing this abbreviated SEM introduction, we move to the most complex term, which includes the linear elasticity tensor. This yields the stiffness matrix, the most expensive operator we will derive. To start we rewrite the stiffness integrand in terms of its inner product terms

$$\nabla \mathbf{w} : \mathbf{T} = \sum_{i,k=1}^3 F_{ik} \frac{\partial w_i}{\partial \xi_k}, \quad (2.9)$$

where

$$F_{ik} = \sum_{j=1}^3 T_{ij} \partial_j \xi_k, \quad F_{ik}^{\alpha\beta\gamma} = F_{ik}(\mathbf{x}(\xi_\alpha, \eta_\beta, \zeta_\gamma)), \quad (2.10)$$

recalling the second-order stress tensor  $\mathbf{T}$  (9 terms) and the  $\xi_k$  terms from the inverse Jacobian, which requires that no mesh elements yield a singular Jacobian (a typical finite-element requirement). In order to evaluate the integral, we need this tensor evaluated at each of the element GLL points

$$\mathbf{T}(\mathbf{x}(\xi_i, \eta_j, \zeta_k)) = \mathbf{C}(\mathbf{x}(\xi_i, \eta_j, \zeta_k)) : \nabla \mathbf{s}(\mathbf{x}(\xi_i, \eta_j, \zeta_k)).$$

Recalling the integral for the stiffness term, we apply the quadrature rule and rearrange terms to produce the stiffness approximation

$$\begin{aligned} \int_{\Omega_e} \nabla \mathbf{w} : \mathbf{T} d\Omega &= \sum_{i,k=1}^3 \int_{\Omega_e} F_{ik} \frac{\partial w_i}{\partial \xi_k} \\ &\approx \sum_{\alpha,\beta,\gamma=0}^{N_\alpha, N_\beta, N_\gamma} \sum_{i=1}^3 w_i^{\alpha\beta\gamma} \\ &\quad \times \left[ \omega_\beta \omega_\gamma \sum_{\alpha'=0}^{N_\alpha} \omega_{\alpha'} J^{\alpha'\beta\gamma} F_{i1}^{\alpha'\beta\gamma} \ell'_{\alpha'}(\xi_{\alpha'}) \right. \\ &\quad + \omega_\alpha \omega_\gamma \sum_{\beta'=0}^{N_\beta} \omega_{\beta'} J^{\alpha\beta'\gamma} F_{i2}^{\alpha\beta'\gamma} \ell'_{\beta'}(\xi_{\beta'}) \\ &\quad \left. + \omega_\alpha \omega_\beta \sum_{\gamma'=0}^{N_\gamma} \omega_{\gamma'} J^{\alpha\beta\gamma'} F_{i3}^{\alpha\beta\gamma'} \ell'_{\gamma'}(\xi_{\gamma'}) \right], \quad (2.11) \end{aligned}$$

where  $\ell'_i = \frac{d\ell}{d\xi}(\xi_i)$  is a result of evaluating  $\nabla \mathbf{w}$ . As with the mass-matrix evaluation, the  $w_i^{\alpha\beta\gamma}$  are the set of Lagrange polynomials, independently zero, such that both sums (over  $\alpha, \beta, \gamma$ , and  $i$ ) drop all but a single term, separating the degrees of freedom. Thus, each row of the stiffness matrix  $\mathbf{K}$ , indexed by  $\alpha, \beta, \gamma$  is defined by the similarly indexed expression in (2.11) containing the degrees of freedom within that element  $\Omega_e$ . Additionally, the expressions for rows that correspond to nodes on an element-element boundary, contain terms from all neighboring elements, a process well known as *assembly*.

Ignoring the boundary and forcing terms, we can write this system as

$$\mathbf{M} \frac{d^2}{dt^2} \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} + \mathbf{K} \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} = 0 \quad (2.12)$$

or, more simply,

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{K}\mathbf{u} = 0, \quad (2.13)$$

where the matrices  $\mathbf{M}$  and  $\mathbf{K}$  are filled by the appropriate terms from (2.8) and (2.11). Furthermore,  $\mathbf{M}$  is diagonal, and  $\mathbf{K}$  is extremely sparse, with each row only containing nonzero entries corresponding to degrees of freedom that are shared by the element or elements that contain the row's respective node.

### Time discretization

The matrix equation (2.13) is finally ready for a time-discretization scheme and, because the mass matrix is diagonal, we can rewrite the system simply as

$$\ddot{\mathbf{u}} = -\mathbf{M}^{-1}\mathbf{K}\mathbf{u}$$

because  $\mathbf{M}^{-1}$  is trivially computable. This form with derivatives of  $\mathbf{u}$  on the left, and linear functions of  $\mathbf{u}$  on the right, allows for the use of explicit time-stepping schemes. Although many schemes are possible, SPECFEM3D currently utilizes the well-known explicit Newmark time-stepping scheme [Krenk, 2006]. A second-order accurate scheme that conserves a discrete quantity (a perturbation of the discrete energy), it can be written as

$$\begin{aligned} \mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_n + \frac{\Delta t^2}{2} \mathbf{a}_n, \\ \mathbf{v}_{n+1/2} &= \mathbf{v}_n + \frac{\Delta t}{2} \mathbf{a}_n, \\ \mathbf{a}_{n+1} &= -\mathbf{M}^{-1}\mathbf{K}\mathbf{u}_{n+1}, \\ \mathbf{v}_{n+1} &= \mathbf{v}_{n+1/2} + \frac{\Delta t}{2} \mathbf{a}_{n+1}. \end{aligned}$$

As can be quickly observed, these time-stepping operations require only vector arithmetic (e.g.,  $\mathbf{u}_n + \Delta t \mathbf{v}_n$ ), with the exception of the computation of  $\mathbf{K}\mathbf{u}_{n+1}$ . Furthermore, these simple vector operations are at the mercy of the memory bandwidth with little room for optimization, whether that is on a standard CPU or a multithreaded GPU.

### 2.3.2 SEM computational structure

One advantage of the explicit Newmark time-stepping scheme shown in the last section is its ability to compute updates in place, that is no additional memory is required beyond the field variables  $\mathbf{u}, \mathbf{v}, \mathbf{a}$ . The SPECFEM3D CPU and GPU code both follow the same basic computational sequence seen in Alg. 1.

---

**Algorithm 1** Newmark time stepping for SEM.

---

**Require:**  $\mathbf{u} = \mathbf{u}_0, \mathbf{v} = \mathbf{v}_0, \mathbf{a} = \vec{0}, \Delta t$ , source term  $\mathbf{S}$ .

- 1: **for**  $t_n = 0, \dots, T_n$  **do**
  - 2:      $\mathbf{u} \leftarrow \mathbf{u} + \Delta t \mathbf{v} + \frac{\Delta t^2}{2} \mathbf{a}$
  - 3:      $\mathbf{v} \leftarrow \mathbf{v} + \frac{\Delta t}{2} \mathbf{a}$
  - 4:      $\mathbf{a} \leftarrow \vec{0}$
  - 5:      $\mathbf{a} \leftarrow \mathbf{K}\mathbf{u} + \mathbf{S}$
  - 6:      $\mathbf{a} \leftarrow -\mathbf{M}^{-1} \mathbf{a}$
  - 7:      $\mathbf{v} \leftarrow \mathbf{v} + \frac{\Delta t}{2} \mathbf{a}$
  - 8: **end for**
- 

Besides the stiffness matrix operation  $\mathbf{K}\mathbf{u}$  in step 5, these steps offer only the low arithmetic intensity of BLAS1 operations, which we've seen to offer only modest speedup opportunities for a GPU, on par with the approximate 3x memory bandwidth gap between a GPU and multicore CPU. The computation of  $\mathbf{K}\mathbf{u}$ , on the other hand, represents a higher arithmetic intensity and thus offers additional opportunities to take advantage of the higher peak floating point performance of the GPU.

#### Stiffness matrix computation

Although there exist efficient sparse-matrix vector multiplication libraries for both CPUs and GPUs, because we know the exact structure of  $\mathbf{K}$ , it is more efficient to implement this operation implicitly as the action of  $\mathbf{K}$  onto  $\mathbf{u}$ . Recall the computation of  $\int_{\Omega_e} \nabla \mathbf{w} : \mathbf{T} d\Omega$  (2.11) that is split into four sums over element nodes indexed by  $\alpha, \beta, \gamma$ , and dimensions indexed by  $i$ . The careful choice of test

function  $w_i^{\alpha\beta\gamma}$  isolates all but the following terms for each index corresponding to a node in each element (or a row in  $\mathbf{K}$ ) allowing us to define a function  $k(\Omega_e, \alpha, \beta, \gamma)$  that computes the contribution to each value of  $\mathbf{a}$  for each element  $\Omega_e$  and node  $\alpha, \beta, \gamma$ :

$$\begin{aligned}
k(\Omega_e, \alpha, \beta, \gamma) = & \omega_\beta \omega_\gamma \sum_{\alpha'=0}^{N_\alpha} \omega_{\alpha'} J^{\alpha'\beta\gamma} F_{i1}^{\alpha'\beta\gamma} \ell'_{\alpha'}(\xi_{\alpha'}) \\
& + \omega_\alpha \omega_\gamma \sum_{\beta'=0}^{N_\beta} \omega_{\beta'} J^{\alpha\beta'\gamma} F_{i2}^{\alpha\beta'\gamma} \ell'_{\beta'}(\xi_{\beta'}) \\
& + \omega_\alpha \omega_\beta \sum_{\gamma'=0}^{N_\gamma} \omega_{\gamma'} J^{\alpha\beta\gamma'} F_{i3}^{\alpha\beta\gamma'} \ell'_{\gamma'}(\xi_{\gamma'}).
\end{aligned} \tag{2.14}$$

In this expression, the field variables  $\mathbf{u}$  defined at each node are embedded within the computation of  $F_{ij}^{\alpha\beta\gamma}$ . However, in this expression, only values of  $\mathbf{u}$  defined within the element contribute, allowing the computation to be structured by element. In fact, the CPU version of SPECFEM3D organizes the updates to  $\mathbf{a}$  as a loop over elements containing a loop over element nodes, the body of which contains the computation of (2.14).

Each of the entries in  $\mathbf{u}, \mathbf{v}$  and  $\mathbf{a}$  corresponds to an element node, some of which are shared between elements. To solve this sharing problem, each entry is given a global index and is referenced using a technique known as *indirect addressing* to map element-local nodes to these global degrees of freedom. This structure yields Alg. 2 outlining the structure of the computation of  $\mathbf{K}\mathbf{u}$  in SPECFEM3D.

---

**Algorithm 2** CPU  $\mathbf{K}\mathbf{u}$  computation in SPECFEM3D.

---

- 1: **for all**  $\Omega_e \in \Omega_h$  **do**
  - 2:     **for all**  $\alpha, \beta, \gamma \in N_\alpha, N_\beta, N_\gamma$  **do**
  - 3:         element-node contribution  $\leftarrow k(\Omega_e, \alpha, \beta, \gamma)$
  - 4:          $i_{\text{global}} \leftarrow \text{global-lookup}(\Omega_e, \alpha, \beta, \gamma)$
  - 5:          $\mathbf{a}(i_{\text{global}}) \leftarrow \mathbf{a}(i_{\text{global}}) + \text{element-node contribution}$
  - 6:     **end for**
  - 7: **end for**
- 

We note that the function  $k(\Omega_e, \alpha, \beta, \gamma)$  embeds a similar global lookup for  $\mathbf{u}$ , and that the in-place sum operation in line 5 is a result of the required finite-element *assembly* due to shared nodes on element boundaries (multiple

elements contain nodes with identical  $i_{\text{global}}$  index values). This choice of CPU implementation drives the mapping to the GPU programming model.

As noted in Sec. 2.2.2, the CUDA threading model exposes and organizes itself in terms of lightweight threads organized in teams called *blocks*. We choose the natural mapping, where the loop over elements is mapped to CUDA blocks, and the loop of nodes is mapped to CUDA threads. The CUDA kernel is written as the body of these loops with the CUDA runtime launching and scheduling the necessary blocks and threads running on each kernel instantiation, outlined in Alg. 3

---

**Algorithm 3** GPU  $\mathbf{Ku}$  computation kernel in SPEC3D

---

```

1:  $\Omega_e \leftarrow \text{blockId}$ 
2:  $\alpha, \beta, \gamma \leftarrow \text{threadId}$ 
3: element-node contribution  $\leftarrow k(\Omega_e, \alpha, \beta, \gamma)$ 
4:  $i_{\text{global}} \leftarrow \text{global-lookup}(\Omega_e, \alpha, \beta, \gamma)$ 
5:  $\mathbf{a}(i_{\text{global}}) \leftarrow \mathbf{a}(i_{\text{global}}) + \text{element-node contribution}$ 

```

---

This mapping of blocks to elements and threads to nodes was done in the original GPU implementation by Komatitsch et al. [2010], but is supported by analysis done by Cecka et al. [2011] and is additionally used by the high-performance DG code for Maxwell's equations in [Klöckner et al., 2009]. With the structure of the kernel set, there are a number of key optimizations, which are critical to the performance of the GPU version, which we outline in the next section.

## 2.4 GPU performance optimizations

Ensuring that the GPU version achieves high performance is critical, due to the high programming and maintenance costs relative to a CPU version. We highlight some of the most critical optimizations including

1. shared-memory caching,
2. fast and correct updates to shared degrees of freedom,
3. optimizing memory operations,
4. asynchronous memory and I/O operations.

The previous section introduced the SEM for elastic wave propagation, including the most expensive operation  $\mathbf{a} \leftarrow \mathbf{K}\mathbf{u}$  which accounts for more than 65% of runtime. As noted, blocks are mapped to elements, and threads to nodes. SPEC3D commonly uses fourth-order finite-element polynomials, which leads to  $N_\alpha \times N_\beta \times N_\gamma = 125$  nodes per hexahedral element, conveniently close to 128. The GPU internally schedules threads in groups of 32 called “warps,” such that when possible, structuring computations around factors of 32 generally produces the highest performance. In the next section we detail the structure of the kernel, including managing cache by hand.

### Shared memory: explicit caching

GPUs, unlike CPUs, require explicit management of cache by making use of explicitly allocated shared memory denoted by the `__shared__` variable prefix. This shared memory is actually on-chip and has near register access speeds (about 40 cycles [Volkov and Demmel, 2008; Wong et al., 2010]) and is visible to all threads within a block. A common CUDA development pattern has each thread fetching a corresponding common value from global memory and storing it in the shared memory cache, followed by the `__syncthreads()` barrier. The function  $k(\Omega_e, \alpha, \beta, \gamma)$  from (2.14) can take advantage of shared memory as certain variables are reused across nodes (and threads). We outline this in Alg. 4 detailing the structure of the CUDA implementation.

---

**Algorithm 4** GPU Kernel  $\mathbf{K}\mathbf{u}$  outlining use of shared memory cache.

---

**Require:** `__shared__ float shared_u[NαNβNγ], shared_JF[NαNβNγ]`

```

1:  $\Omega_e \leftarrow \text{blockId}$ 
2:  $\alpha, \beta, \gamma \leftarrow \text{threadId}$ 
3:  $i_{\text{global}} \leftarrow \text{global-lookup}(\Omega_e, \alpha, \beta, \gamma)$ 
4:  $\text{shared\_u}_{x,y,z}[\text{threadId}] \leftarrow \mathbf{u}_{x,y,z}(i_{\text{global}})$ 
5: __syncthreads()
6:  $\text{shared\_JF}_{(x,y,z),j}[\text{threadId}] \leftarrow J_{(x,y,z),j}^{\alpha,\beta,\gamma}(\Omega_e) F_{(x,y,z),j}^{\alpha,\beta,\gamma}(\text{shared\_u}_{x,y,z})$ 
7: __syncthreads()
8:  $\mathbf{a}_{x,y,z}(i_{\text{global}}) \leftarrow \mathbf{a}_{x,y,z}(i_{\text{global}}) +$ 
9:  $\omega_\beta \omega_\gamma \sum_{\alpha'=1}^{N_\alpha} \omega_{\alpha'} \ell'_\alpha(\xi_{\alpha'}) \text{shared\_JF}_{(x,y,z),1}(\alpha', \beta, \gamma) +$ 
10:  $\omega_\alpha \omega_\gamma \sum_{\beta'=1}^{N_\beta} \omega_{\beta'} \ell'_\beta(\xi_{\beta'}) \text{shared\_JF}_{(x,y,z),2}(\alpha, \beta', \gamma) +$ 
11:  $\omega_\alpha \omega_\beta \sum_{\gamma'=1}^{N_\gamma} \omega_{\gamma'} \ell'_\gamma(\xi_{\gamma'}) \text{shared\_JF}_{(x,y,z),3}(\alpha, \beta, \gamma')$ 

```

---

The first use of shared memory is the storage of the field vector  $\mathbf{u}_{x,y,z}$ , which is

used by all the threads in step 6 by the function  $F_{(x,y,z),j}^{\alpha,\beta,\gamma}(\mathbf{u}_{x,y,z})$ . Step 6 also stores this result into shared memory, for use again in steps 9,10, and 11 by all threads in this block. This use of shared memory as a cache to share computed values between threads is critical to the efficiency of this CUDA kernel and follows the standard practices in the *CUDA Programming Guide*.

### Mesh coloring vs. atomic operations

The careful reader will notice that step 8 in Alg. 4, where the update to the acceleration  $\mathbf{a}_{x,y,z}(i_{\text{global}})$  occurs, is not thread-safe on nodes which are shared between elements. Figure 2.3 gives an example of a 2D rectangular mesh with 4 elements. The nodes at  $a, c, d$ , and  $e$  all share and receive updates from two elements, where node  $b$  is shared by all four.

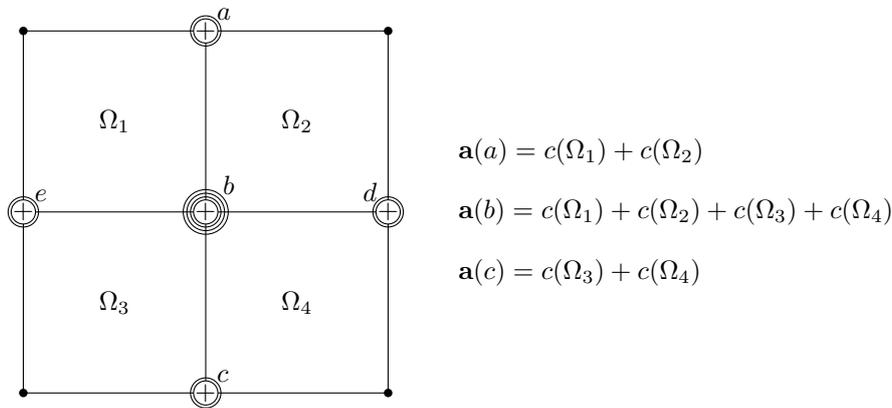


Figure 2.3. Example 2-D mesh highlighting shared nodes (degrees of freedom) between elements. Shared nodes are marked with a “+” and have one additional circle per shared element, such that the center node at “ $b$ ” has three additional circles. The corner nodes are not shared and thus not labeled.

Because the nodal value can be incremented concurrently by multiple threads, this results in a race condition unless properly guarded. With so many threads in flight when using CUDA, incorrect updates occur frequently and must be handled appropriately. The CPU-only MPI-parallel version avoids this by computing the intrapartition element contributions in serial, with MPI synchronization applied to perform the sum on shared nodes that span two partitions or more. In a multithreaded shared-memory model, such as OpenMP on the CPU or CUDA on the GPU, one must guarantee that this assembly operation on a shared node is done one at a time. Two possible ways of doing this are by using hardware sup-

ported synchronization primitives, such as atomic operations, or, alternatively, by using mesh coloring.

Atomic memory operations offer a convenient mechanism for avoiding these race conditions, however, possibly with a significant overhead. The Fermi generation GPUs (e.g., Tesla X2090) used to conduct some of the experiments within this chapter and Rietmann et al. [2012] had improved atomic performance over their predecessor (e.g., Tesla C1050), but still presented a performance loss overall. Fortunately NVIDIA was able to further optimize the atomic operation pipeline in their Kepler generation GPUs, making any alternative solutions obsolete. However, given the speed at which technology changes, we present our mesh-coloring solution for future accelerator technologies that might have less optimized atomic operations.

Mesh coloring [Komatitsch et al., 2009, 2010] solves the concurrent assembly problem by algorithmically guaranteeing that element contributions to shared nodes are calculated and updated at different points in time. A mesh-coloring algorithm takes the elements in a partition  $\Omega$ , and creates  $K$  disjoint subsets  $\Omega_k$ , such that the

$$\bigcup_k^K \Omega_k = \Omega$$

and that the elements in each  $\Omega_k$  do not share any boundaries or points. This partitioning guarantees that elements can be computed in parallel and that their boundary nodes are updated safely without synchronization primitives, such as `atomicAdd()` in CUDA. Note that mesh coloring adds a serial loop on all the colors in the solver, with CUDA kernel calls for each color, and is thus only a good option when the number of colors obtained is small from a given finite-element mesh, which is fortunately always the case in practice. An example coloring with 4 colors is given in Fig. 2.4. Note that no two sets  $\Omega_k$  have elements that touch.

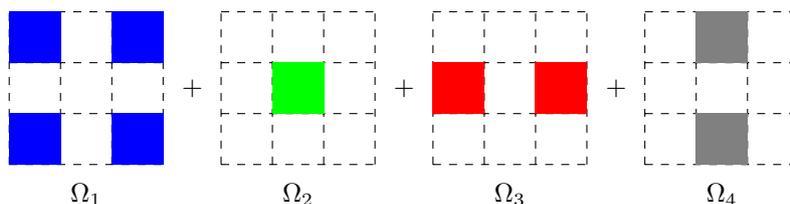


Figure 2.4. 2D rectangular mesh with 6 elements using 4 colors to guarantee that no elements touch.

In general, an optimal coloring is one with the fewest possible colors; however, a coloring with a balanced number of elements/color could have better

performance due to better load balancing. Currently two coloring methods are available within SPECfem3D:

- *First Fit (FF)*: A simple greedy algorithm that always chooses the first possible color [Krivelevich, 2002; Komatitsch et al., 2009]. This produces a coloring with a near-optimal coloring in terms of numbers of colors, but is often quite unbalanced due to its greedy nature.
- *Droux*: A balanced, but nonoptimal coloring following a “least used” (LU) coloring principle, which is based on a modified implementation of Droux [1993].

The greedy coloring method uses fewer colors than the Droux algorithm, but the balance of elements per color is not perfect. The number of colors and their balancing, however, does not seem to affect performance as noted in the top panel of Figure 2.5. In addition to performance, another factor in the choice of coloring algorithm is the time required for the coloring itself. The FF coloring requires 15x less time than Droux to generate. Given the identical performance at runtime for the hexahedral meshes in this study, we recommend using the simplest greedy algorithm (FF), if only to save time and complexity in preprocessing.

In Figure 2.5 we further compare the two coloring methods against the standard atomic update along with the noncoloring, nonatomic benchmark, which by definition produces incorrect results as mentioned above, but proves useful as a comparison because it provides an upper bound on the performance of any such operation. Both coloring options produce performance similar to this incorrect method, illustrating that the overhead for using a colored mesh is negligible and that any reasonable coloring method can and should be used.

Cecka et al. [2011] determined that doubling the number of colors yields less than a 10% performance loss, and our own experiments depicted in Figure 2.5 show that neither the number of colors nor the balance of elements per color have a strong effect on performance. In fact, the incorrect, noncoloring, but also nonatomic, code performs only slightly better than the coloring versions, indicating both that assembly via mesh coloring can achieve near peak performance, and that any reasonable coloring is a good coloring. In the end, however, the improved performance of `atomicAdd()` on Kepler generation cards allows us to make use of atomic operations to ensure correctness without sacrificing performance, reducing the complexity of the final code.

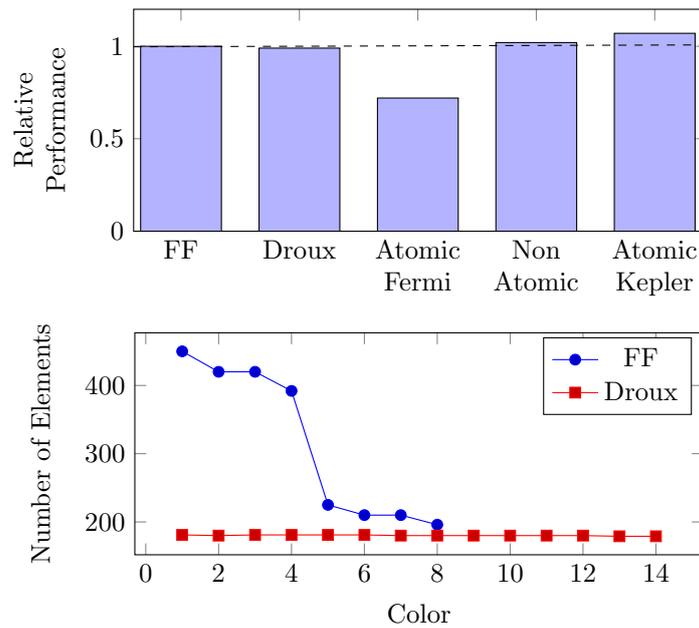


Figure 2.5. (top) The relative performance of two GPUs on a 303,116 element mesh. The first two bars represent two separate coloring algorithms. The third bar represents a noncolored version, which uses CUDA’s `atomicAdd()` operation running on a Fermi generation GPU. The fourth bar uses neither coloring nor atomic updates for incorrect results but high performance. The final bar shows the relative improvement of atomic operations on Kepler generation GPUs. (bottom) The distribution of colors from a small test mesh to demonstrate the balance of colors between FF and Droux. In this example, the FF algorithm uses the optimal number of colors for a regular hexahedral mesh (8), but with an uneven distribution. Droux requires more colors, but distributes them evenly.

### Coalesced, aligned, and texture memory transfers

As noted in Sec. 2.2, many algorithms, including explicit time-stepping finite-element methods are limited by memory bandwidth. Taking full advantage of the GPU’s memory bandwidth requires the accessed memory to be word aligned such that a 32-thread warp can coalesce its memory reads and writes. Some of the variables that make up the computation of  $\mathbf{Ku}$  are unique to each element and node and were 128-padded to make use of these coalesced memory transfers. As discussed previously, the fields  $\mathbf{u}$  and  $\mathbf{a}$  are stored in terms of global degrees of freedom and use a lookup variable via *indirect addressing* to access their values. This lookup of variable indexes is sometimes called a *gather and*

*scatter* operation and is well known for its effect on performance. As discussed in Markall et al. [2013], one possible option is to duplicate shared nodes in the field variables  $\mathbf{u}$  and  $\mathbf{a}$  allowing us to also pad them to groups of 128 (by element). This technique, however, requires an additional step to recombine the duplicated values. Our experiments indicated that this additional step destroyed the speedup gains realized by the duplicated and padded arrays.

Beyond this, it was discovered that using global memory for certain element constants yields a 10% performance gain compared with using the cached constant memory. Constant memory and its cache produce better performance when many threads access a single constant memory location; however, we have many threads accessing multiple locations in constant memory, which instead produced slower performance.

In order to gain a further 10%, the displacement ( $\mathbf{u}$ ) and (possibly) acceleration ( $\mathbf{a}$ ) vectors are bound to texture memory [Komatitsch et al., 2010]. The texture cache does not maintain coherency within a kernel launch, leaving the acceleration variable vulnerable to incorrect results. Mesh coloring does guarantee that each acceleration value updated during the stiffness assembly is only read and written by a single thread. The texture cache is flushed between kernel launches, ensuring the correctness of the next shared-node acceleration update. However, Kepler’s improved atomic pipeline picks up the slight performance lost by coloring, enough to only use the texture cache for the  $\mathbf{u}$  variable.

### Asynchronous CPU-GPU memory transfers

Meshes for simulations of interest can easily range from  $10^6$  to  $10^9$  degrees of freedom, such that memory and node allocations must be tailored to each specific application. While the global-scale code SPEC-FEM3D\_GLOBE uses analytical, hard-coded meshes for the full Earth, SPEC-FEM3D can work with arbitrary, user-provided hexahedral meshes generated with a program such as the CUBIT Mesh Generation Toolkit,<sup>9</sup> offering modeling flexibility to the scientist. Partitioning of the unstructured mesh is accomplished by the SCOTCH graph partitioning library, resulting in high-quality domain decompositions that ensure low surface to volume ratio, edge-cut minimization, and optimized load balancing.

The spatial partitioning has the requirement that, at each time step, the local elements that border another partition, or *halos* (also called “outer” elements), need to be exchanged amongst nearest neighbors, introducing a loose global coupling. The reference CPU version of SPEC-FEM3D tries to hide these MPI

---

<sup>9</sup>CUBIT, originally developed by Sandia National Laboratories, is now commercially developed and supported by csimsoft under the name Trellis

communications necessary at each time step by overlapping the computation of  $\mathbf{Ku}$  with the necessary synchronizations between mesh-partition neighbors. To implement this exchange, each time step is split into an outer and inner phase and asynchronous MPI communications are used to send messages containing the updates of the outer elements to neighboring partitions. Thus the solver algorithm follows the pseudocode in Alg. 5.

---

**Algorithm 5** SPEC-FEM3D code structure
 

---

```

for  $t = 1, N_{\text{STEPS}}$  do
  Time-step update ()
  for phase=outer,inner do
    Stiffness Assembly (phase)
    Absorbing Boundaries (phase)
    Source Forcing (phase)
    MPI-Communications (phase) // sent asynchronously
  end for
  Time-step finalize ()
  Calculate Seismograms ()
end for

```

---

In this pseudocode listing, the “outer” and “inner” phases correspond to elements on the partition boundary halo and the inner, nonhalo region, respectively. For example, *Stiffness Assembly (phase=outer)*, calculates the stiffness update only for elements in the outer halo region. The MPI communications for the outer phase are nonblocking, which allows the inner-element stiffness, boundary, and source contributions to be computed while the MPI communications are being completed concurrently when the MPI implementation and cluster is setup to support this ability. This overlap of computation and communication results in excellent scalability of the code to large processor counts.

In order to recreate these communication-hiding features in the GPU version, additional care was required. For the GPU version, the synchronization at each time step requires a GPU→CPU memory copy before the MPI communications can begin. By using page-locked pinned memory for the MPI buffer, we can send the required updates to the CPU from the GPU asynchronously using the `cudaMemcpyAsync()` function. The computational timeline in Fig. 2.6 outlines our strategy to hide most of the communications required by the GPU version. The “outer” elements that share a boundary or node with elements in a different partition (shaded in blue) are computed first. An asynchronous memory copy is launched once these outer elements are finished, which is directly followed by

the launching of the CUDA kernel to compute the inner elements.

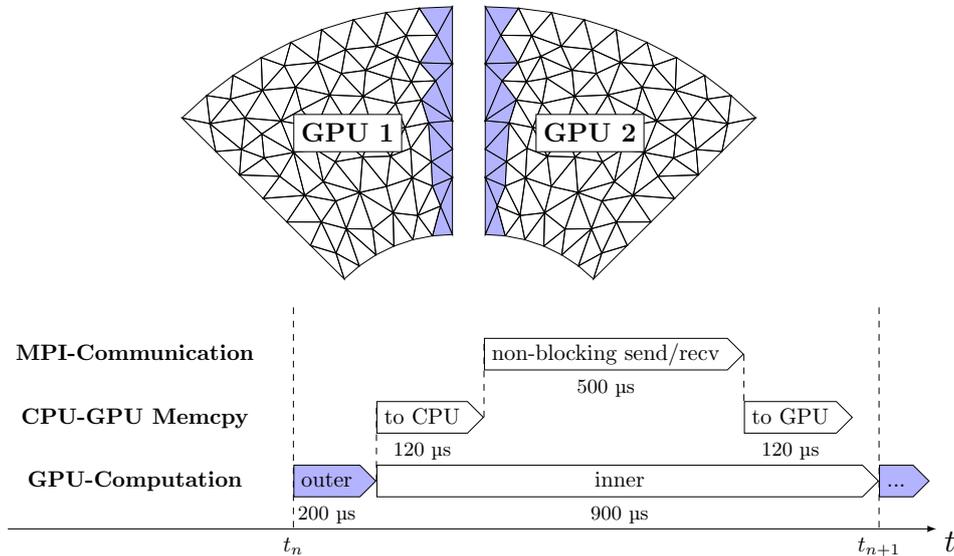


Figure 2.6. A view of the computational time line, outlining our overlapping strategy. The shaded region contains elements in the “outer” domain, which share a boundary, an edge, or a point with another partition and are computed first. The time line elements are not displayed exactly to scale, but their measured timing for a 190,000 element mesh on 16 GPUs is included directly below each, respectively.

Both the asynchronous memory copy and CUDA kernel launches are non-blocking, which allows the main thread to actively wait for the memory copy to finish. Upon completion of the copy to the CPU, nonblocking MPI communications can be started. As soon as both the inner element computation and MPI have finished, the MPI-transferred updates are asynchronously copied back to the GPU and added to their respective elements. The next time step can begin once this update finishes, repeating the same process. If the ratio of inner to outer elements is high enough ( $>10,000$  elements per GPU), the code will have effectively hidden both the memory copy and communication required to synchronize the two partitions. The experiments that revealed the timings in Fig. 2.6 tell us that this communication hiding paradigm via asynchronous memory transfers and nonblocking MPI is especially critical to the scaling efficiency of this GPU version.

## 2.5 Performance Evaluation

For the applications in seismology motivating the work in this thesis, we setup experiments simulating an earthquake, where the faster we achieve a target time, the higher the performance. In general, for these types of simulations, the cost is only dependent on the number of elements in the mesh and not on the earthquake source or receivers. We calculate the performance by measuring the time loop for 1,000 time steps. The number of FLOP per element for a single time step was counted manually (41,833), and used to estimate floating point performance for all meshes used in this study as

$$\begin{aligned} \text{Performance (GFLOP/s)} = & \\ & \left( \frac{41,833 \text{ FLOP/element/time step}}{\text{time-stepping runtime (s)}} \right) \times \left( \frac{\# \text{ elements} \times K \text{ steps}}{1} \right) \\ & \times \left( \frac{1 \text{ GFLOP}}{1 \times 10^9 \text{ FLOP}} \right). \end{aligned}$$

Note that we ignore the time required to set up and initialize memory on the CPU and/or GPU, which for shorter time iterations would lead to a significant portion of the time-to-solution performance, especially for GPUs. However, for almost all user applications, time iterations of several tens of thousands of time steps are required, making such an effect of memory initialization negligible in practice.

For our benchmarks, we will plot the parallel efficiency of the simulation which is the speedup  $S = T_s/T_r$ , where  $T_r$  is the measured runtime and  $T_s$  the serial runtime (or 2-GPU run time in our case), divided by the number of parallel processes  $N$  (or in our case the number of nodes). In an attempt to model how well our code scales, we further compare the simulation runtime  $T_r$  against a simple model based on Amdahl's law:

$$T_\alpha = \alpha \frac{T_s}{N} + (1 - \alpha)T_s, \quad (2.15)$$

where  $T_\alpha$  is the effective total run time and the parameter  $\alpha$  indicates the fraction of code which was parallelized effectively.

As noted in the coloring analysis, technology has moved fast during the last years, obsoleting certain results. The GPUs and GPU-clusters originally used to develop this work are no longer available (e.g., Fermi-generation X2090 GPUs have been replaced by Kepler-generation K20X GPUs). Nevertheless, we present the original published results [Rietmann et al., 2012] in addition to new results from a newer cluster.

To run our simulations, we use four different Cray systems:

- a Cray XK6/XK7 system (Tödi)<sup>10</sup> located at the Swiss National Supercomputing Center (CSCS) that features 176 nodes, each equipped with a 16-core AMD Opteron 6272 CPU, 32 GB of DDR3 memory, and was originally designed with one NVIDIA Tesla X2090 GPU (Fermi/XK6). These GPUs were later replaced with one NVIDIA Tesla K20X GPU (Kepler/XK7) with over 5 GB of available GDDR5 memory;
- a Cray XE6 system (Monte Rosa) also at CSCS to run our CPU scaling experiments on, with 2×AMD Opteron 6272 2.1 GHz 16-core CPUs, and 32 GB memory per compute node. Each Opteron 6272 core shares its floating point hardware with a second core, limiting the linear scalability of SPECfem3d to only 8 cores per socket. Monte Rosa features a total of 1,496 nodes for a total of 47,872 cores, with a theoretical peak performance of 402 TFLOP;
- a Cray XK6 system (Titan),<sup>11</sup> which was in development at the Oak Ridge Leadership Computing facility, where we conducted our largest GPU simulations on 896 nodes. It comprises the same XK6 compute nodes as used at CSCS;
- a Cray XC30 system (Piz Daint) with nodes comprised of a single 8-core Intel XEON E5 CPU<sup>12</sup> and a single NVIDIA K20X GPU detailed in Sec. 2.2. It has 5,272 nodes with peak realized performance of 6.2 PFLOP, making it the world's #6 supercomputer as listed by TOP500 in June 2014.

### 2.5.1 Benchmarking GPU versus CPU simulations

Previous GPU speedup results such as Klöckner et al. [2009] present performance results comparing a single GPU against a single CPU core. As the work in this thesis is done in the context of computational seismology, we are primarily interested in the performance of the software when run using supercomputing resources, which very often contain two multicore CPUs (Rosa) or a single CPU with a single GPU (e.g., Titan, Tödi, Piz Daint). The modern Cray XC30 nodes

---

<sup>10</sup>This cluster was updated in late 2012 from the XK6 configuration with X2090 Fermi GPUs to a XK7 configuration with K20X Kepler GPUs.

<sup>11</sup>This development cluster was upgraded to XK7 nodes (each with a single K20X GPU) and expanded to include 18,688 nodes and was the world's fastest supercomputer as rated by Top500 in November 2012.

<sup>12</sup>*Sandy Bridge* microarchitecture.

can be equipped with either two 8-core Intel CPUs or the configuration in Piz Daint with a single 8-core CPU and the single GPU. We compare our performance results on a node-to-node basis e.g., a single node of Rosa (with 32 MPI processes per node, one per core) and a single node of Tödi (one process per node for each GPU). For the newer results, we compare CPU and GPU results on Piz Daint, however as we do not have ready access to CPU-only XC30 nodes, we are forced to compare CPU results using only 8-cores per node (one process per core) against the GPU results (one process per node for each GPU). Ideally these results would be compared using 16-core CPU nodes, but we can already estimate these results by using more nodes, as will be noted in the results.

We believe this node-to-node comparison is relatively fair, from both a cost and power perspective. From a scientific and HPC perspective, cost should prove to be the final deciding factor as hybrid architectures move from development to production at many sites worldwide. Ideally, a project would outline their total computational requirements for a specific scientific milestone and work with a supercomputing center to detail the total cost for the project in terms of hardware and power usage. This discussion, however, should also consider the higher programmer and maintenance effort required by CUDA programming, which may be a significant factor in cost decisions. Additionally, the long-term power and cooling for a system should also be considered, i.e., costs associated with actually running the system for a few years once it is installed.

## 2.5.2 Weak scaling

In order to test that the code can scale to large problem sizes, we conduct so-called weak scaling tests. In these weak scaling benchmarks our test model<sup>13</sup> has a variable resolution such that we can fill  $\sim 75\%$  of each GPU's memory, keeping the load per MPI process constant while increasing the number of total processes. Ideally, the time-to-solution will remain constant as we increase the problem size by adding CPUs or GPUs. Each XK6 or XC30 GPU node has 413,952 elements (3.9 GB of GPU memory each) where, on the XE6 or XC30, each CPU node (32-cores or 8-cores, respectively) has 422,400 elements. Figure 2.7 depicts both CPU and GPU performance for an increasing problem size. In this case, the CPU and GPU codes scale perfectly, that is their performance graphs are linear on a log-log scale (twice the nodes, twice the performance). We also plot the parallel efficiency, i.e., the measured speedup  $T_s/T_r$  divided

---

<sup>13</sup>The weak-scaling mesh models earthquake simulations in southern California using the `xmeshfem3D` tool.

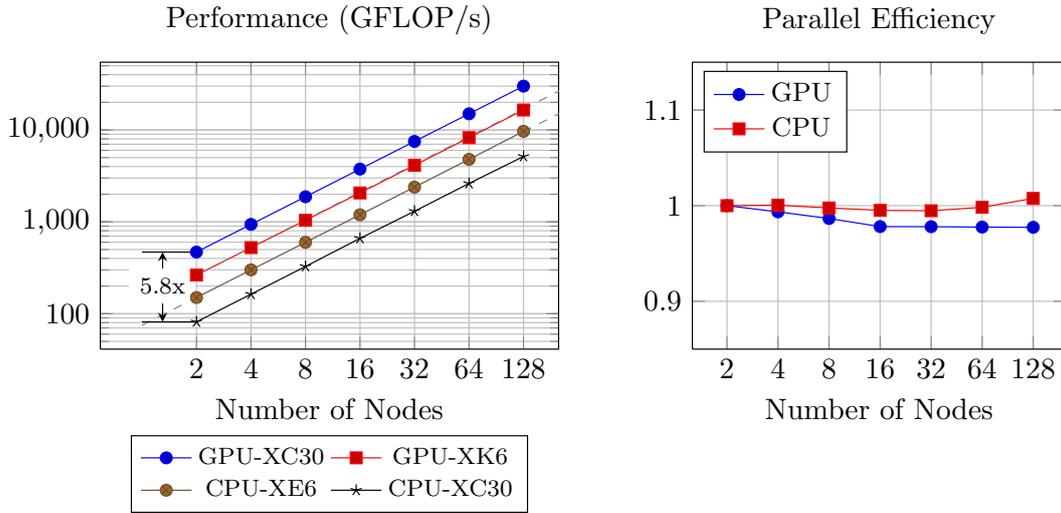


Figure 2.7. Weak scaling performance results comparing GPU (XC30,XK6) and CPU (XE6,XC30) nodes each with 400K elements per node. (left) Performance results showing a speedup factor of  $\sim 5.8x$  between the newer XC30 CPU and GPU nodes. (right) Parallel efficiency for CPU and GPU nodes.

by the number of nodes  $N$ . Parallel efficiency is normalized to 2 XE6 and XK6 nodes, respectively. Comparing weak scaling performances between GPU and CPU nodes, we see a speedup factor of 5.8x between the XC30 GPU and CPU nodes. We note that two XC30 CPU nodes (one 8-core Intel CPU each) provide about 10% additional performance over the older XE6 nodes (two 16-core AMD CPUs each). The newer Kepler-generation K20X GPUs are about 75% faster than the older Fermi-generation 2090X GPUs.

The parallel efficiency for both GPU and CPU simulations is excellent, scaling within 98% of ideal runtimes across all benchmark sizes. This indicates that asynchronous message passing is nearly perfect in hiding the network latency on these systems. Note that this becomes more of a challenge for smaller mesh sizes, where the percentage of outer elements increases compared to inner elements. As both the GPU and CPU simulations scale very well with an increasing problem size, we can expect to run geographically large simulations with high resolution on either GPUs or CPUs very efficiently.

### 2.5.3 Strong scaling

The weak scaling experiments demonstrate that when each CPU and GPU has sufficient work (i.e., elements), the performance scales extremely well. To com-

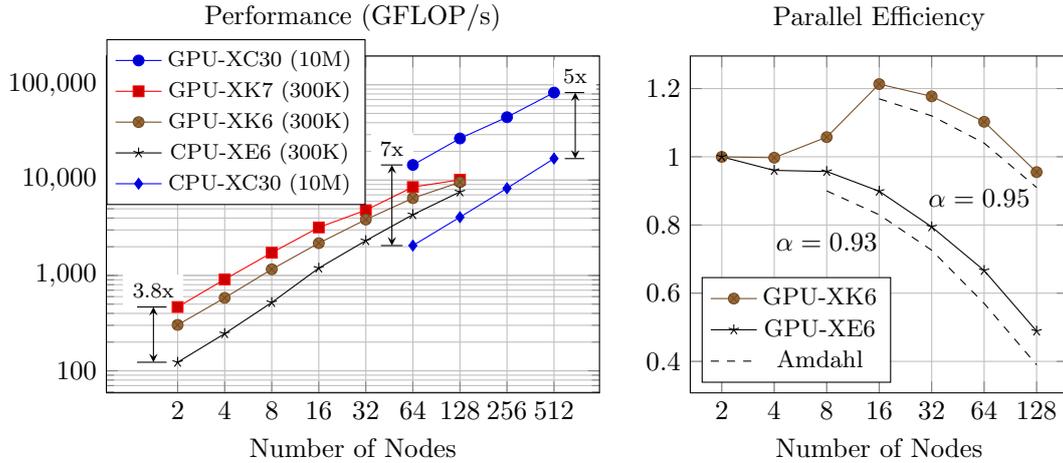


Figure 2.8. Strong scaling results on XK6 GPU, XK7 GPU, XE6 CPU, XC30 CPU+GPU nodes: (Left) Performance results for 300K and 10M element meshes, showing an initial speedup factor of 3.8x between XK7 and XE6, and 7x between XC30 GPU and CPU nodes. (Right) Parallel efficiency for strong scaling results. We note the greater than 1 initial CPU parallel efficiency, a result of improved cache efficiency.

plement these experiments, we choose several fixed size meshes and perform strong scaling experiments, where the problem size is fixed, and additional CPUs or GPUs are added in order to reduce the time-to-solution. We again compare the performance using both the older (and now unavailable) XK6 GPU nodes, XE6 CPU nodes, and newer K20x-based XC30 nodes. The results, seen in Fig. 2.8, are run on a smaller 300K element mesh using 2–128 nodes (XK6/XE6), and a larger 10M element mesh using 64–512 nodes (XC30).

The scaling results show a 3.8x GPU speedup over the CPU version running on full CPU XE6 nodes and a 7x speedup against half-CPU XC30 nodes, using the newer K20X GPUs, which represents a 1.5x speedup over the older Fermi-generation GPUs. We note that the CPU version scales superlinearly, as seen by the 120% parallel efficiency at 16 nodes. As the mesh is split over more processors, the degrees of freedom per process decreases and consequently improves cache utilization, an effect that is not as effective in the GPU version as the caching mechanism is designed differently.

In these strong scaling benchmarks, Amdahl’s law (2.15) plays a critical role. The following components of the simulation are inherently serial, or only represented in a subset of partitions, causing load imbalance or serial bottlenecks that affect the GPU version more strongly than the CPU version:

**MPI Communications (1-5%):** Although we attempt to hide the MPI message sending and receiving with the (inner) **Ku** kernel, the GPU's high performance makes it more difficult for the MPI systems to keep up. Experiments on *Piz Daint* revealed widely varying latencies for these messages, between 400  $\mu$ s and 5 ms. At 70K elements per GPU, the inner **Ku** kernel required 8.4 ms, which adequately covered the MPI latency. However, with 2K elements per GPU, the kernel only took 200  $\mu$ s, whereas the MPI communications still took 400–600  $\mu$ s. In this case, MPI communications are no longer covered by the computation and create the scaling inefficiencies seen in Fig. 2.8. If we assume that the computation of **Ku** takes 7x more time on the CPU (e.g.,  $200 \times 7 = 1.4$  ms), the minimum latency (e.g., 400  $\mu$ s) will still be hidden by computation.

**Station recording (1% (300  $\mu$ s) per 10 stations):** Seismic stations set to record velocity and displacement on the mesh surface are often not evenly distributed in the mesh domain. Additionally, their values must be copied from the GPU to the CPU at each time step. Obviously the number of stations and their distribution across GPUs plays a role here.

**Source (<1%):** The earthquake is often modeled as a point source localized to a single element in a single partition.

For the strong scaling experiments, for each curve we estimated  $\alpha$ , the percentage of the computation that is being parallelized:

Mesh	$\alpha$	Reason(s) (via profiling)
300K	0.93	MPI, GPU-CPU Transfers, Stations
CPU 300K	0.95	MPI
10M	0.93	MPI, GPU-CPU Transfers, Stations

As one can see, the CPU and GPU version share a similar parallelization percentage, and that this percentage remains constant across mesh sizes, if we measure  $\alpha$  as shown in Fig. 2.8. In general, as the partition sizes become small, and the ratio of partition surface to volume becomes larger, MPI communications become the major efficiency bottleneck. However, the number of stations and their distribution can play a significant role in certain scenarios.

## 2.5.4 Large application test

A common example of seismic forward modeling lies in the quantification of ground shaking due to earthquakes. Such simulations are important for comparing synthetic seismograms based on current knowledge of the structural region

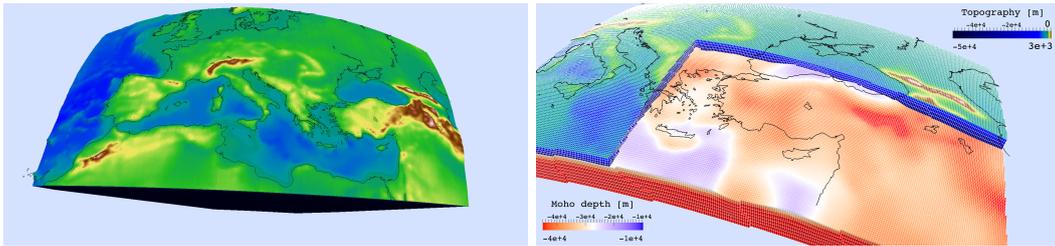


Figure 2.9. A 3D unstructured crust and mantle model in Europe including 19 million spectral elements, honoring surface topography (left) and the undulating crust-mantle Moho interface (right).

against observations and lead to better assessments in seismic hazard analysis. We focus on earthquakes in Turkey, a country that is subjected to high seismic hazard. Our settings simulate ground motions resulting from significant fault ruptures close to Istanbul.

We construct a spectral element mesh that honors surface topography (Fig. 2.9, left panel) and the undulating crust-mantle interface (called the “Moho” interface in geophysics; Fig. 2.9, right panel), spanning from Turkey to Portugal, North Africa to Scandinavia, and down to a depth of 1,500 km (i.e., in the mid-mantle of the Earth). The mesh contains about 19 million spectral elements, leading to 3.8 billion degrees of freedom and collective runtime GPU-memory occupation of 260 GB with 290 MB per GPU for 896 XK6 nodes. We model three earthquake scenarios and run the simulations on 896 and 448 Titan nodes to test scalability for this realistic example. On 896 nodes, the solver requires 1682 s to complete the necessary 75,000 time steps for 1500 s seismograms (i.e., almost real-time simulation), yielding 35 TFLOP/s of floating point performance. For the same setup on 448 GPUs, we have 26 TFLOP/s of performance.

In order to evaluate the resulting seismic waves, seismograms were recorded at stations in Turkey and Europe including a synthetic grid of stations encompassing Istanbul. The recording stations in this experiment were heavily clustered, in particular, due to the grid of stations around Istanbul, causing a load imbalance across the 896 partitions of the 19M element mesh seen in Fig. 2.9. Repeating the 896-GPU simulation with only a single station required less than half the time, yielding 78 TFLOP/s of performance. This type of problem is not as pronounced in the CPU version because the individual mesh partitions are 32x smaller, better distributing the stations among processors.

The goal of this simulation was to model a ground motion investigation in Istanbul produced by a hypothetical magnitude 7.3 earthquake located in the Sea of Marmara fault region, 20 km southwest of Istanbul. As part of the analy-

sis, a  $21 \times 21$  grid of stations was placed on a 60 km grid surrounding Istanbul, in order to evaluate vertical and horizontal seismic ground shaking for the city following the study done in Pulido et al. [2004]. Because the mesh and corresponding Earth model was still relatively coarse relative to the city of Istanbul, the ground motion recordings did not provide a particularly interesting result. Using the ability to include localized high resolution mesh elements to model such local phenomena from Ch. 3 could enable such types of analyses.

## 2.6 Roofline Model for SPECfEM3D

Performance and scaling benchmarks can only give us relative performance results, compared to a CPU version, and do not yield insight into how close the code is to peak performance. The roofline model, however, introduced in Sec. 2.2 can give us an idea of how close we are.

Instead of using the raw peak-performance and bandwidth numbers, we continue our analysis with synthetic benchmark numbers. Using a workstation-class NVIDIA K20c GPU, we measure  $B = 150$  GB/s using a stream benchmark and  $F = 2.5$  TFLOP/s based on a large SGEMM benchmark. This is compared to 50 GB/s and 375 GFLOP/s for an Intel 8-core CPU. The algorithm implemented by SPECfEM3D can be characterized by 2 stages, dominated by the computation of the stiffness kernel matrix multiplied by the displacement vector ( $\mathbf{Ku}$ ). This stiffness kernel operation has an estimated  $AI = 5$ , and when combined with the time-stepping operations the total  $AI = 3$ . This full-program  $AI$  is relatively low and indicates that the program will be mostly limited by memory bandwidth.

We visualize the roofline performance model for our K20c in Fig. 2.10 along with the benchmarks from our newly developed SPECfEM3D GPU code. We see that we are achieving more than 50% of the modeled peak performance in both the stiffness kernel update and the complete time-stepping loop. The performance achieved is a result of the many optimizations discussed in Sec. 2.4, however additional profiling yields several operations that are difficult to fix using the current approach including the following:

**Register Pressure:** The stiffness operation ( $M^{-1}Ku$ ) for elastic wave propagation requires a large number of intermediate variables, stressing the CUDA register allocator and reducing the occupancy of the GPU compute units, creating a reduction in performance. This reduction in performance already occurs for the simplest elastic medium with an isotropic medium and no attenuation. Additional code and variables for any additional physical properties such as general anisotropy or attenuation only make the

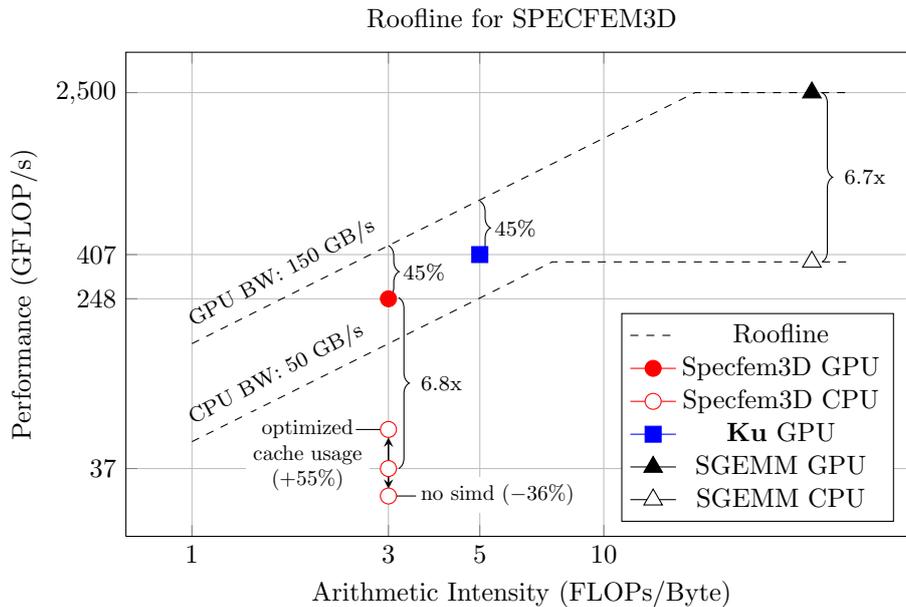


Figure 2.10. Roofline performance model for SPECfEM3D measured on Tesla K20c and 8-core Intel CPU. Peak performance is SGEMM benchmark, and memory bandwidth is STREAM benchmark. The circles represent the performance of the full time-stepping loop and the square (GPU only) shows the performance of the stiffness kernel **Ku**. Additionally we have highlighted how optimizations of memory and vector operations affect the CPU version; in particular, we modeled improved cache utilization and the effect of removing SIMD operations.

register problem worse and may require us to rethink the layout of the elastic stiffness kernel.

**Gather-Scatter:** As mentioned, the reading and writing of the field variables **u** and **a** results in an unaligned gather-scatter memory access pattern that significantly reduces effective memory bandwidth. This also reduces the CPU version’s cache utilization, and is simply an unavoidable property of the SEM spatial discretization. Improved layout of the field variables in memory according to spatial access patterns (e.g., using space-filling curves [Bader and Zenger, 2006]) would improve both GPU and CPU performance by reducing the number of global memory accesses. Estimates based on the strong scaling performance indicate that improving cache utilization would increase performance for the GPU version by 15% and for the CPU version by 55%.

The CPU comparison for the overall SPECfEM3D performance indicates that

the CPU code should be performing at a higher level, closer to the bandwidth limit achieved by the GPU version. The CPU version has had some initial optimizations to take advantage of cache within the element-local  $\mathbf{Ku}$  update; however, as seen by the cache utilization improvement in the strong scaling benchmarks, the layout of mesh elements and degrees of freedom is not optimal. By calculating the degree of parallelization  $\alpha$  from Amdahl's law (2.15) for the CPU and GPU versions, we can estimate how improved memory locality (due to the smaller partition size) contributes to the performance. Based on the strong scaling results in Fig. 2.8, improved field-memory layout can yield an estimated 55% more performance for the CPU version and 15% for the GPU version, reducing the example speedup from 6.8x to 5x.

We also performed an experiment with the SIMD (AVX256) turned off (flag `-no-simd` using Intel's `ifort` fortran compiler), which resulted in a 36% loss in performance for the CPU version. If we assume the worst case, where the loss of AVX256 (8-wide vectors with single precision) will result in a factor 8 slowdown in peak performance, the (non linearized) roofline model (2.1) predicts a performance loss of only 66% due to the low  $AI$  of the full time-stepping loop. Nonetheless, it may be interesting to implement the  $\mathbf{Ku}$  method using Intel's experimental SIMD compiler (`ispc`) [Pharr and Mark, 2012] to see the effect of possibly improved vectorization over what the Intel and Cray Fortran compilers can achieve.

### 2.6.1 Emerging architectures outlook

As seen by the nearly 7x speedup seen by the GPU version over an 8-core Intel CPU, parallel computing architectures such as GPUs can have a significant impact. For 6 months, the world's fastest supercomputer was ORNL's Titan, which was comprised of the previously described Cray XK7 nodes, each with a single AMD 16-core CPU and a K20X NVIDIA GPU. It was surpassed by the TIANHE-2, a supercomputer in China equipped with 16 thousand nodes, each with two 8-core Intel CPUs and *three* Intel Xeon Phi accelerators. These Xeon Phi (MIC) accelerators contain 60+ simplified cores, much like a GPU. Each MIC can execute AVX512 instructions, which are capable of performing vector operations of width 16 in single precision. These 16-width vectors are potentially comparable to the 32-thread "warps" used on the GPU to coalesce the individual GPU threads. These accelerators also have a similar memory bandwidth to an NVIDIA Tesla GPU. Based on our roofline model and other experiments, a SPECFEM3D version supporting these Xeon Phi processors should be comparable to our GPU version.

The planned replacement for Titan at ORNL, will be comprised of IBM POWER9 CPUs and a future NVIDIA GPU architecture called VOLTA, which has 3D stacked memory for a reported 4x higher memory bandwidth (1 TB/s).<sup>14</sup> Based on the roofline model, this should directly increase the performance by a similar factor. However, the MPI system latency will also have to be decreased or optimized in order to maintain the strong scaling performance of the GPU version presented here.

## 2.7 Adjoint Tomography

The large-scale modeling application detailed in Sec. 2.5.4 is a typical use of so-called *forward* modeling, which uses an existing Earth velocity and density model to examine how waveforms propagate through an existing domain. Looking at the global-scale simulations done by Komatitsch et al. [2003] or our own simulation in the introduction, the simulated and real recorded seismograms compare quite well, despite the relative simplicity of the 1D (radial) velocity model used. Improving this velocity model brings us to the fields of *tomography* or *seismic imaging*, which are framed in terms of *inverse* modeling. Using adjoint methods we can iteratively improve the fit between the simulated seismograms and the seismograms recorded after real earthquakes around the globe. These techniques, however, are incredibly expensive, but can make use of our existing forward solvers (such as SPECSEM3D) with only minimal modification. This pushes us to utilize techniques such as GPU computing to try and drastically improve the throughput of these simulations, in order to make these seismic imaging projects more manageable and foster experimentation.

### 2.7.1 Adjoint methods overview

Just as we briefly introduced the SEM for the elastic wave equation in Sec. 2.3.1, we will briefly introduce the adjoint method, with a focus on the computationally expensive aspects of the algorithm.

The method employed by SPECSEM3D is outlined by Tromp et al. [2004]. As noted, the goal is to minimize the differences between recorded (real) multi-component waveform data  $\mathbf{d}(\mathbf{x}_r, t)$  recorded at  $N$  stations  $\mathbf{x}_r$ , and the corresponding simulated (synthetic) data  $\mathbf{s}(\mathbf{x}_r, t, \mathbf{m})$  for a given earth model  $\mathbf{m}$ . Thus we define the least-squares misfit functional to measure the fit between simu-

<sup>14</sup><http://www.anandtech.com/show/8727/nvidia-ibm-supercomputers>.

lated and real data,

$$\chi(\mathbf{m}) = \frac{1}{2} \sum_{r=1}^N \int_0^T \|\mathbf{s}(\mathbf{x}_r, t, \mathbf{m}) - \mathbf{d}(\mathbf{x}_r, t)\|^2 dt. \quad (2.16)$$

In order to reduce the misfit, we need to find the misfit perturbation due to a model perturbation, yielding a Fréchet derivative

$$\delta\chi = \sum_{r=1}^N \int_0^T [\mathbf{s}(\mathbf{x}_r, t, \mathbf{m}) - \mathbf{d}(\mathbf{x}_r, t)] \cdot \delta\mathbf{s}(\mathbf{x}_r, t, \mathbf{m}) dt, \quad (2.17)$$

where  $\delta\mathbf{s}$  is the perturbation in the displacement field due to a model perturbation  $\delta\mathbf{m}$ . By computing the appropriate best model perturbation (gradient), we can update the model iteratively, using standard (gradient-only) nonlinear optimization tools and algorithms such as low memory BFGS [Wright and Nocedal, 1999]. Our model  $\mathbf{m}$  is defined by the density  $\rho(\mathbf{x})$  and fourth-order elastic tensor  $c_{ijklm}$  (which contains wave velocities). This adjoint procedure allows us to derive the so-called *waveform misfit kernels*  $K_{\rho, c_{ijklm}}$ , which specify the first-order descent direction that can be used to update the model  $\mathbf{m}$ :

$$K_{\rho}(\mathbf{x}) = - \int_0^T \rho(\mathbf{x}) \mathbf{s}^{\dagger}(\mathbf{x}, T-t) \cdot \partial_t^2 \mathbf{s}(\mathbf{x}, t) dt, \quad (2.18)$$

$$K_{c_{ijklm}}(\mathbf{x}) = - \int_0^T \epsilon_{jk}^{\dagger}(\mathbf{x}, T-t) c_{ijklm}(\mathbf{x}) \epsilon_{lm}(\mathbf{x}, t) dt, \quad (2.19)$$

where  $\mathbf{s}^{\dagger}$  and  $\epsilon^{\dagger}$  are the adjoint displacement field and adjoint strain field tensors, respectively. The adjoint fields ( $\mathbf{s}^{\dagger}, \epsilon_{ij}^{\dagger}$ ) are defined by the same elastic wave equation defining the standard fields  $\mathbf{s}$  and  $\epsilon$ , however with the adjoint source

$$f_i(\mathbf{x}, t) = \sum_{r=1}^N [s_i(\mathbf{x}_r, T-t) - d_i(\mathbf{x}_r, T-t)] \delta(\mathbf{x} - \mathbf{x}_r) \quad (2.20)$$

which is the time-reversed difference between the simulation and data at each recording station. The adjoint also requires zero final conditions (as opposed to initial conditions), i.e.,  $\mathbf{s}^{\dagger}(\mathbf{x}, T) = 0$ . This has the result that the adjoint is simulated in reverse time, backwards from  $T$ , as also seen in the kernel definition (2.18) with  $\mathbf{s}^{\dagger}(\mathbf{x}, T-t)$ .

Following a standard nonlinear optimization scheme, we update our model using these misfit kernels and a corresponding step size  $\alpha$  chosen by line search, trust region, or other methods

$$\mathbf{m}_{n+1} = \mathbf{m}_n + \alpha(K_{\rho, c_{ijklm}}), \quad (2.21)$$

where  $\alpha$  guarantees that  $\chi(\mathbf{m}_{n+1}) < \chi(\mathbf{m}_n)$ , i.e., the new model reduces the waveform misfit. We note that methods such as conjugate gradient or BFGS can improve the search direction  $K_{\rho, c_{jklm}}$  in addition to providing a predicted step length  $\alpha$ . Thus we have an iterative process that will converge to a new model  $\mathbf{m}$ , which produces a forward simulation that produces waveforms that are closer to the original data (in a least-squares sense).

The particular computational challenges for the model update are twofold:

1. The computation of  $K_{\rho}$  and  $K_{c_{jklm}}$  require the simultaneous evaluation of the forward fields with the adjoint fields. This requires either the storage of the full forward field  $\mathbf{s}(\mathbf{x}, t)$  for all time steps (usually infeasible), or the backwards reconstruction of the field starting at  $t = T$ , and integrating backwards in time following the adjoint field. This reduces the required storage to only saving the field values at absorbing boundaries to reconstruct the full forward field (in backwards time).
2. We defined both the misfit kernel  $K_{\rho, c_{jklm}}$  and misfit functional  $\chi(\mathbf{m})$  in terms of a single earthquake source. In practice, however, we compute the misfit functional and kernel as the sum of many sources in order to produce adequate spatial coverage between sources and stations. This increases the number of simulations to compute each misfit and kernel by the number of sources used. We could, of course, compute the model updates for one source at a time, but each source misfit and kernel can be computed independently, drastically increasing the available parallelism, allowing us to take advantage of ever increasing supercomputing resources.

### 2.7.2 Adjoint costs

For common tomography applications, it is usual to consider hundreds of sources to ensure good spatial coverage. A common tomography example will contain a database using hundreds of earthquakes and their seismogram traces. Thus to compute a single misfit kernel we must simulate one forward simulation per source in order to define the adjoint sources. This is followed by the second step of evaluating the misfit kernel, requiring two equivalent simulations (adjoint + forward fields). Each source kernel is summed to build a full source kernel, in order to update the model  $\mathbf{m}$ .

We also extended the standard earthquake-source adjoint functionality using new noise-based techniques. To enhance seismic resolution in locations with little seismicity such as Western Europe, one can conveniently exploit information by cross correlating stacked ambient noise (generated by, e.g., ocean-continent

interactions) between two seismographs [Shapiro et al., 2005]. Such cross correlations deliver signals reminiscent of surface-wave propagation from one seismograph to the other. In principle, one can thus use any pair of seismic stations as a combination of source and receiver, enabling seismic tomography without earthquakes to see a dramatic growth in seismic coverage and resolving power.

This *noise tomography* approach was recently added to SPEC-FEM3D [Tromp et al., 2010]. It requires an additional first step, such that we require two forward simulations to define the adjoint source, and the adjoint with forward (in backwards time) simulation in order to create the misfit kernels. Thus, each kernel requires four effective simulations (one more than the standard source approach).

If we consider 20–30 model iterations and 150 stations, we expect 12,000 – 18,000 equivalent forward simulations, which works out to 5–10M CPU hours, or more than 300K node hours. This additionally assumes minimal all-source misfit computations, making the optimization algorithm design more difficult due to the cost of evaluating even a single misfit. Thus, this project was tasked to help reduce these costs by speeding up the forward and adjoint simulations with GPU computing.

### 2.7.3 I/O optimizations

Both the standard adjoint and noise tomography approaches require a significant amount of I/O. As mentioned, in the standard adjoint approach, the first step involves simulating a standard earthquake source, where we save the field at the boundaries for all time steps. These boundaries are read and injected into the simulation in reverse time for the second step where the misfit kernel is generated.

The noise tomography algorithm requires saving the forward wavefield at the mesh surface in the first step. This is followed by a second step where this surface wavefield is used to generate the appropriate forward field. In this second step the absorbing boundaries are saved just like in the standard adjoint approach. In the third step, the saved boundaries are read and injected in reverse time to reconstruct the forward wavefield (in backward time) for the misfit kernel.

Once the initial GPU implementation of the adjoint functionality was completed, it has been critical to benchmark and profile the full adjoint application on a production cluster with challenging I/O conditions. Hidden by the performance of local disks and caching, a critical performance flaw appeared when run using centralized “scratch” file systems. These file systems scale to very high performance, but have different throughput and latency characteristics than a

local workstation.

### Concurrent file output

Previous experiments with the CPU version to utilize threads to concurrently write the boundaries to disk at each time step did not yield an important speedup to overall performance. Because the CPU version is almost 7x slower, the disk output was only a small fraction of runtime. However, the increased speed of the GPU version means that this small fraction can add up to 50% to the runtime. We implemented an efficient threaded boundary writing routine, which attempts to overlap the standard simulation with each absorbing boundary field time-step output. This reduced the disk-writing overhead to 10–25%, increasing our overall GPU speedup of step 1 to 5x.

### Pathological file access

The second step, however, provided a critical optimization opportunity, for both the CPU and GPU versions of the code. As noted, to implement the misfit kernels (2.18)–(2.19), we simulate the forward field in reverse time using the saved boundaries to fully contract the fields. This means that we read the saved absorbing boundaries from the end to the start, essentially in reverse. For most file-reading libraries (e.g., `fread` on linux), reading a file in reverse presents a pathological performance case, yielding the worst possible performance. This is because these libraries achieve their performance through buffered caches, which are buffering in the wrong direction (file forwards instead of backwards). This can be compared to reading a large matrix in the wrong direction, where cache lines are loaded orthogonal to successive memory loads.

For our particular example on the cluster *Piz Daint*, the system call to write each field to disk (forwards) took 1.8 ms to complete, which makes it less than 5% of each time step’s runtime. The corresponding read, in reverse order, took **30–1150 ms**, an enormous slowdown that dominates the overall runtime. Clearly, solving this file-reading problem is critical to both an efficient CPU version, and to maintaining the hard-won performance gains of the GPU version. In order to solve this problem, several solutions were implemented, including the following:

1. **Read entire file to memory:** A good possibility depending on the size of the mesh, the number of nodes, and the number of required time steps. However, there are many cases where this is not feasible due to memory constraints.

2. **Threaded prebuffering:** Prebuffer the file using a separate thread that (hopefully) remains ahead of the simulation.

Both solutions solve this file input problem, either by avoiding reading the file backwards, or through careful application of threading and buffered file reads.

### Threaded reading

This solution provides the lowest memory footprint, but does bring the highest implementation complexity. We present a diagram of the implementation in Fig. 2.11.

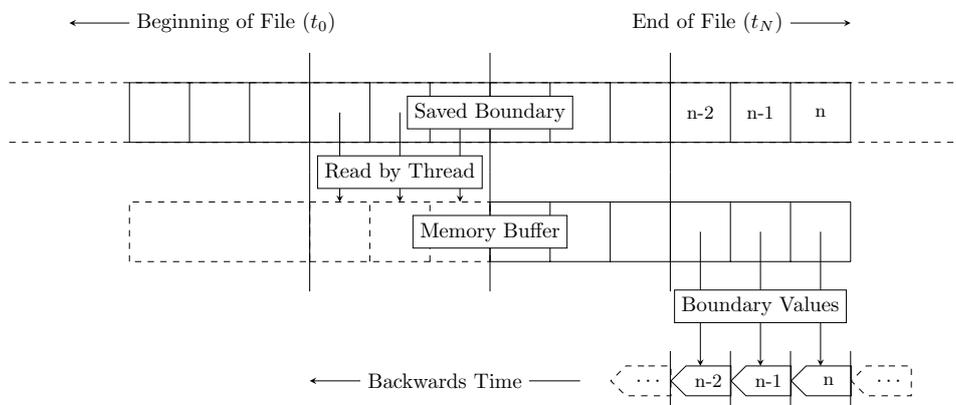


Figure 2.11. Threaded file read prebuffering timeline diagram. Each horizontal block corresponds to the saved field at a single time step  $t_n$ . We note that the file is read by a separate thread in multi-time-step blocks, backwards from the end of the file to the start. The simulation copies the field values from the finite-length memory buffer. The thread tries to stay several steps ahead of the simulation.

The GPU version, as currently written, does not take advantage of the available cores and we thus try to utilize CPU threading where possible to help hide I/O behind computation. The threaded-reading routine maintains a thread which reads from the absorbing boundary in multi-time-step chunks (3 steps in the figure, but 50+ in practice) into the memory buffer, amortizing the file seeking and reading overhead over multiple steps. In the figure, each block contains the full field information at each time step in each block ( $n, n-1, n-2$ ). The memory buffer is large enough to contain multiple sets of steps (4 in the figure, but 10+ in practice), allowing the thread to (hopefully) stay multiple steps ahead of the simulation (in backwards time). The simulation utilizes the preloaded boundary values from the memory buffer, freeing their space once read. Not

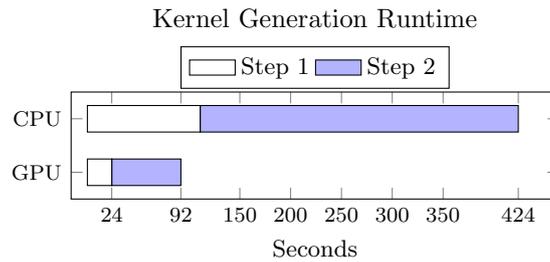


Figure 2.12. Runtime comparison of the two steps for a single adjoint sensitivity kernel. All three steps were run on 16 XC30 nodes for the CPU and GPU versions. The runtimes shown measure 5000 steps of the time step loop only. The step 2 duration is approximately 2.8x longer than the step 1 duration for both CPUs and GPUs.

shown in the figure, the memory buffer is circular, allowing the thread to reuse buffer blocks once read by the simulation.

Both file-reading solutions presented reduce the file-reading overhead to less than 10% of the total runtime (for GPUs, less for CPUs). We will now consider a benchmark example to highlight the new performance of the new adjoint I/O subsystem and GPU version of SPEC-FEM3D.

### 2.7.4 Adjoint performance benchmarks

In order to highlight the performance of our GPU and I/O optimizations, we conducted experiments on a mesh of 300,000 elements, designed to cover Europe at an appropriate spatial resolution. Because the discussed costs of tomography are so high, every effort is made to make the mesh as small as possible, allowing the *many* simulations to be run quickly. For this benchmark, we compare 16 nodes on *Piz Daint*, with 128 cores and 16 GPUs. We ran the simulation for 5000 time steps, but a full tomography application would likely require more, to allow the waves to fully propagate through the medium for all sources. A comparison of the runtime between the CPU and GPU is shown in Fig. 2.12

Profiling indicates that step 1 loses up to 25% of runtime due to the absorbing boundary output, and 10% (GPU) or 1% (CPU) for the adjoint step when prebuffered or threaded I/O were used. We note from Fig. 2.12 that the adjoint step (step 2) requires 2.8x more time than the forward step. Given that step 2 must simulate both the forward and the adjoint fields, in addition to computing the misfit kernels, this is expected. Most important for this thesis, the overall GPU speedup is 4.6x, meaning that it takes **4.6x less time** to compute each

source misfit kernel.

These tomography benchmarks, where the misfit Kernel is computed, bring together the GPU optimizations from the previous sections, but demonstrate critical I/O optimizations done for this project. Benchmarks run on different clusters before the threaded reading was implemented indicated a 10–100x slowdown in step 2.

## 2.8 Conclusion

This chapter detailed our work on a GPU-enabled version of SPEC-FEM3D, which includes facilities to do both forward and inverse modeling via adjoint-based sensitivity kernels. The major takeaway is the relative performance chart in Fig. 2.13 where we can see the performance across the AMD, and Intel CPUs and NVIDIA GPUs seen in this chapter.

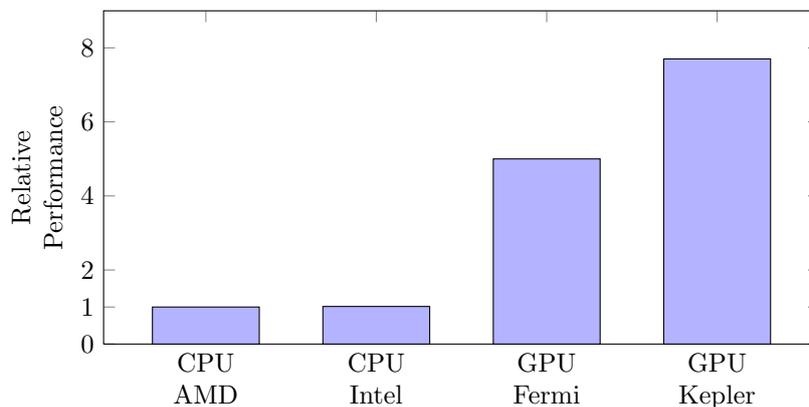


Figure 2.13. Relative performance across CPUs and GPUs evaluated in this thesis. Performance relative to the CPU AMD measurement, a single-socket 16-core Interlagos CPU, which is compared to the 8-core Intel CPU, the X2090 Fermi-generation GPU, and the K20X Kepler-generation GPU.

This factor 7–8x increase in performance is the result of focused design decisions and careful optimizations based on GPU profiling and performance experiments. Starting with Sec. 2.1 we outlined the GPU architecture, as compared to the CPU architecture, highlighting how the GPU’s relative pipeline simplicity allows for increased floating point performance. Using the Roofline model we outlined the performance differences between CPUs and GPUs, which was revisited in Sec. 2.6. This roofline model exposed that the CPU version is actually lagging behind the GPU version, if we assume that both GPU and CPU

should achieve a similar fraction of the roofline-modeled peak performance at SPECFEM3D's estimated  $AI$ .

After introducing the SEM used by SPECFEM3D, we outlined its mapping to GPU hardware in Sec. 2.3.2, the critical design step that allows the GPU version to achieve the 7x performance speedup we have seen. Of course, the mapping simply provides the foundation and the many optimizations in Sec. 2.4 provide the final performance. This includes the use of shared-memory cache, coalesced memory operations, and mesh coloring, which Kepler's new atomic pipeline made obsolete.

Validating these performance optimizations were the strong scaling and weak scaling experiments. The key takeaway from these experiments, especially when strong scaling, is that the GPU's speed puts additional pressure on the MPI subsystem, especially its latency. The expensive  $\mathbf{Ku}$  operation happens over 7x faster on the GPU, giving the MPI messaging system 7x less time to complete its operations before the process stalls waiting on MPI. As we move into ever-faster architectures, it will be important the MPI subsystem keep up with the computational improvements to prevent unavoidable MPI bottlenecks reducing the hard fought performance gain by the accelerator software, running on NVIDIA GPUs, Intel MICs, or another future high-speed architecture.

We also saw the additions made to the adjoint functionality of SPECFEM3D in Sec. 2.7, allowing the GPU version to compute misfit kernels. As seen, the structure of this computation uncovered a performance-killing flaw in the I/O system in SPECFEM3D. The reading of the absorbing boundaries at each time step happens from the end of the file to the front, and on the systems evaluated here, was very slow (50–100x slower than without). By introducing an asynchronous version that uses CPU threads to both read and write these boundaries we nearly eliminated the overhead due to this necessary I/O. With these optimized operations the GPU version is nearly 5x faster than the CPU version for the given experiment. For a real-world scenario, this means we can complete a full tomography, usually requiring months of computation, within several weeks. Or, conversely, we can double the horizontal resolution (4x more elements) and remain comfortably within the same computational profile.

However, the improvements in this chapter are inherently limited to the chosen wave-propagation model (elastic vs. acoustic), spatial discretization (SEM), and time-stepping scheme (explicit Newmark). For example, a finite difference spatial discretization, although numerically comparable, can usually be implemented more efficiently, providing higher performance and possibly lower implementation complexity. These algorithmic choices can often yield as much or more improvement seen by the GPU version. In the next chapter of this thesis

we derive a new form of Newmark local time-stepping scheme that yields additional performance improvements. Significantly, this algorithmic optimization can utilize the GPU version from this chapter, making the two performance gains *multiply* for much higher performance than each factor alone can achieve.



## **Chapter 3**

# **Newmark Local Time Stepping**

## 3.1 Introduction

Efficiently simulating wave propagation at large scales has many important scientific and industrial application domains. In the field of seismology, simulating seismic waves resulting from an earthquake or other seismic source is an important modality used to better understand the Earth's interior structure and dynamic behavior. Many applications in both forward and inverse modeling have been pushing limits of traditional HPC resources for many years. Much of the optimization work in this field is focused on improving the implementation of standard algorithms, which can have bottlenecks that only better algorithm design can remove. Transformative improvements to simulation performance will likely require a coupling of algorithmic, hardware, and software improvements.

In general, the motivating application drives the choice of spatial discretization including a handful of comparable methods, including finite differencing, continuous and discontinuous finite elements, and finite volumes. Finite-element and volume methods are able to use meshes that easily adapt to the spatial domain — some elements can be small where small features are required, and large where large features are needed. However, when a standard explicit time-stepping scheme is used, these small elements require a small time step, enforcing a small time step everywhere in the mesh.

For Newmark and other explicit time-stepping schemes, any local areas of mesh refinement will reduce the global time step thus reducing the efficiency of the method. There are many reasons for local-mesh refinement, but to list a few we see in practice:

1. external and internal 3D geometry/topography;
2. localized small-scale physics (faults/oceans);
3. mesh-generator difficulties (especially for hexahedral elements).

Without a good way to avoid the performance hit of localized refinement, the application scientist usually reduces the scale of the simulation to fit within a computational budget. By working on this algorithmic bottleneck, we accelerate current applications, but also enable future work that was previously infeasible.

In this chapter we propose a method of local time stepping (LTS) that allows the time step to be adapted to the mesh-local spatial resolution. One implementation of LTS is the ADER-DG scheme proposed by Dumbser et al. [2007], which allows for multirate or LTS in each element in a near optimal fashion. However, this method only works for a discontinuous Galerkin (DG) spatial discretization, which is not always desired. Further work using a DG method was

done by Godel et al. [2010] was able to show a multirate algorithm working on GPUs for Maxwell's equation. For continuous finite elements, Madec et al. [2009] presents an energy-conserving LTS scheme working across a fluid-solid interface.

Missing thus far has been an LTS scheme and corresponding high performance implementation focused on continuous Galerkin finite elements such as the spectral element method (SEM). To simplify the development of an LTS variant of the Newmark time-stepping scheme for a SEM, we embrace the framework developed in Diaz and Grote [2009]. They were able to prove and demonstrate optimal convergence and stability properties for second and fourth order leapfrog methods. Using and extending the framework, we derive an LTS variant of the Newmark time-stepping scheme with additional considerations for the SEM, absorbing boundary conditions, and multiple refinement levels. This analysis additionally allows us to implement the scheme with minimal LTS overhead for both CPUs and GPUs in SPEC-FEM3D, building on our results from the previous chapter. We note that the multilevel nature of this LTS scheme presents a load-balancing bottleneck when run across many processors in parallel, which we solve in Ch. 4.

In the next section we will introduce our LTS-Newmark method, with attention to absorbing boundaries, new extensions for many refinement levels, and attention to implementation for continuous finite elements, which is unique as compared to the discontinuous finite-element methods mentioned previously. We follow with numerical experiments to show the stability and convergence properties of the scheme as well as details on the implementation in three dimensions using SPEC-FEM3D.

## 3.2 Newmark Time Stepping for Wave Propagation

As seen in Ch. 2, we are particularly interested in the elastic wave equation, which is commonly used to model seismic wave propagation through the Earth's crust and mantle. If we recall, for the displacement  $\vec{u}$  with  $x$ ,  $y$ , and  $z$  components we have

$$\rho(\vec{x}) \frac{\partial^2 \vec{u}}{\partial t^2} - \nabla \cdot \mathbf{T}(\vec{x}, t) = f(\vec{x}_s, t), \quad (3.1)$$

which is subject to a boundary condition with  $\hat{\mathbf{r}} \cdot \mathbf{T} = 0$  on the free surface with outward normal  $\hat{\mathbf{r}}$ . The vertical and lower boundaries usually use a type of absorbing boundary condition. The stress  $\mathbf{T}(\vec{x}, t)$  is related to the displacement

gradient  $\nabla \vec{u}$  via Hooke's constitutive law

$$\mathbf{T}(\vec{x}, t) = \mathbf{C}(\vec{x}) : \nabla \vec{u}(\vec{x}, t), \quad (3.2)$$

where  $\mathbf{C}$  is the fourth-order elasticity tensor with 21 independent parameters in the fully anisotropic case.

As derived in Ch. 2, we utilize the SEM, a higher-order continuous Galerkin finite-element method. After the standard finite-element treatment, we can write the viscoelastic wave equation in the following matrix form, for the discretized degrees of freedom  $\mathbf{u}$ :

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{K}\mathbf{u} = \mathbf{F}. \quad (3.3)$$

Recall that GLL collocation points combined with the appropriate quadrature rule allow us to achieve an exactly diagonal mass matrix. Thus, the previous equation can be rewritten in a form that allows for an explicit time-stepping scheme,

$$\begin{aligned} \ddot{\mathbf{u}} &= -\mathbf{M}^{-1}(\mathbf{K}\mathbf{u} - \mathbf{F}) \\ &= \mathbf{B}\mathbf{u} + \tilde{\mathbf{F}}, \end{aligned} \quad (3.4)$$

utilizing the fact that the diagonal mass matrix inverse  $\mathbf{M}^{-1}$  is computed trivially allowing  $\mathbf{B}$  to fully represent the spatial discretization. Each degree of freedom (DOF)  $\mathbf{u}$  represents a GLL point (node) on each element, with some DOFs shared between elements. To finalize the discretization of (3.4), we must choose a time-stepping method.

### 3.2.1 Newmark time stepping

We now introduce the parameterized Newmark time-stepping scheme:

$$\begin{aligned} \mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_n + \left(\frac{1}{2} - \beta\right)\Delta t^2 \mathbf{a}_n + \beta \Delta t^2 \mathbf{a}_{n+1}, \\ \mathbf{a}_{n+1} &= \mathbf{B}\mathbf{u}_{n+1} + \tilde{\mathbf{F}}, \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \Delta t \left((1 - \gamma)\mathbf{a}_n + \gamma \mathbf{a}_{n+1}\right). \end{aligned} \quad (3.5)$$

We choose the parameters values  $\gamma = \frac{1}{2}$  and  $\beta = 0$  in order to obtain second-order accuracy and an explicit scheme [Krenk, 2006; Komatitsch et al., 2010],

$$\begin{aligned} \mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_n + \frac{\Delta t^2}{2} \mathbf{a}_n, \\ \mathbf{a}_{n+1} &= \mathbf{B}\mathbf{u}_{n+1} + \tilde{\mathbf{F}}, \\ \mathbf{v}_{n+1} &= \mathbf{v}_n + \Delta t \left(\frac{1}{2} \mathbf{a}_n + \frac{1}{2} \mathbf{a}_{n+1}\right), \end{aligned} \quad (3.6)$$

which is currently used in several well-known spectral element implementations [Fichtner et al., 2009; Nissen-Meyer et al., 2008; Peter et al., 2011] and is generally very popular. In order to derive an LTS version of this scheme, we will first derive the Newmark scheme from first principles. Consider our second-order system (with  $\mathbf{F} = 0$ )

$$\ddot{\mathbf{u}} = \mathbf{B}\mathbf{u},$$

which can be split into a system of first order equations,

$$\begin{aligned}\dot{\mathbf{u}} &= \mathbf{v}, \\ \dot{\mathbf{v}} &= \mathbf{B}\mathbf{u}.\end{aligned}\tag{3.7}$$

We can rewrite this system using an integral formulation

$$\begin{aligned}\mathbf{u}(t_n + \xi\Delta t) &= \mathbf{u}(t_n) + \int_{t_n}^{t_n + \xi\Delta t} \mathbf{v}(s) ds, \\ \mathbf{v}(t_n + \xi\Delta t) &= \mathbf{v}(t_n) + \int_{t_n}^{t_n + \xi\Delta t} \mathbf{B}\mathbf{u}(s) ds.\end{aligned}\tag{3.8}$$

This system is still exact, and in order to approximate the system we must approximate the integrands  $\mathbf{v}(s)$  and  $\mathbf{B}\mathbf{u}(s)$ . The choice of time-stepping scheme tells us how to approximate these integrands. In deriving the Newmark scheme from this integral formulation, we utilize a staggered stepping method, where  $\mathbf{u}(t)$  and  $\mathbf{v}(t)$  are stepping on staggered temporal grids, whereby we can utilize the same midpoint integration approximation for both variables:

$$\begin{aligned}\mathbf{v}(t_n + \frac{1}{2}\Delta t) &= \mathbf{v}(t_n - \frac{1}{2}\Delta t) + \int_{t_n - \frac{1}{2}\Delta t}^{t_n + \frac{1}{2}\Delta t} \mathbf{B}\mathbf{u}(s) ds \\ &\approx \mathbf{v}(t_n - \frac{1}{2}\Delta t) + \Delta t \mathbf{B}\mathbf{u}(t_n), \\ \mathbf{u}(t_n + \Delta t) &= \mathbf{u}(t_n) + \int_{t_n}^{t_n + \Delta t} \mathbf{v}(s) ds \\ &\approx \mathbf{u}(t_n) + \Delta t \mathbf{v}(t_n + \frac{1}{2}\Delta t).\end{aligned}\tag{3.9}$$

In order to write this staggered scheme more succinctly, we introduce the notation

$$\begin{aligned}\mathbf{v}_{n+\xi} &= \mathbf{v}(t_n + \xi\Delta t), \\ \mathbf{u}_{n+\xi} &= \mathbf{u}(t_n + \xi\Delta t),\end{aligned}$$

such that we can write (3.9) as the complete Newmark scheme

$$\begin{aligned}\mathbf{v}_{n+\frac{1}{2}} &= \mathbf{v}_{n-\frac{1}{2}} + \Delta t \mathbf{B}\mathbf{u}_n, \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_{n+\frac{1}{2}}.\end{aligned}\tag{3.10}$$

By introducing the variables

$$\begin{aligned}\mathbf{a}_n &= \mathbf{B}\mathbf{u}_n, \\ \mathbf{a}_{n+1} &= \mathbf{B}\mathbf{u}_{n+1}, \\ \mathbf{v}_n &= \mathbf{v}_{n-\frac{1}{2}} + \frac{1}{2}\Delta t \mathbf{B}\mathbf{u}_n,\end{aligned}\tag{3.11}$$

we can rewrite  $\mathbf{u}_{n+1}$  as

$$\begin{aligned}\mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_{n-\frac{1}{2}} + \frac{1}{2}\Delta t^2 \mathbf{B}\mathbf{u}_n + \frac{1}{2}\Delta t^2 \mathbf{B}\mathbf{u}_n \\ &= \mathbf{u}_n + \Delta t \mathbf{v}_n + \frac{1}{2}\Delta t^2 \mathbf{a}_n.\end{aligned}\tag{3.12}$$

By taking an additional half-step from  $\mathbf{v}_{n+\frac{1}{2}}$  we have

$$\mathbf{v}_{n+1} = \mathbf{v}_{n+\frac{1}{2}} + \frac{1}{2}\Delta t \mathbf{B}\mathbf{u}_{n+1},$$

which can be reorganized to eliminate the half steps in  $\mathbf{v}$ ,

$$\begin{aligned}\mathbf{v}_{n+1} &= \mathbf{v}_{n-\frac{1}{2}} + \frac{1}{2}\Delta t \mathbf{a}_n + \frac{1}{2}\Delta t \mathbf{a}_n + \frac{1}{2}\Delta t \mathbf{B}\mathbf{u}_{n+1} \\ &= \mathbf{v}_n + \frac{1}{2}\Delta t \mathbf{a}_n + \frac{1}{2}\Delta t \mathbf{a}_{n+1}\end{aligned}\tag{3.13}$$

with (3.11), (3.12), and (3.13) yielding the system in (3.6).

This Newmark scheme, like any explicit time-stepping scheme used for wave propagation, is only conditionally stable, dependent on an appropriate choice of time step to keep the solution from “blowing up,” i.e., becoming unstable. As usual, the time step  $\Delta t$  is limited by the Courant-Friedrichs-Lewy (CFL) condition, defined by the following inequality,

$$\Delta t \leq C_{\text{CFL}} \min_{i \in \Omega_h} (\tilde{h}_i), \quad \tilde{h}_i = \frac{h_i}{c_i},\tag{3.14}$$

where each element  $i \in \Omega_h$  is an element of the finite-element mesh  $\Omega_h$ , and  $h_i$  and  $c_i$  correspond to the size and material velocity of the  $i$ th element.  $C_{\text{CFL}}$  is the CFL constant, which can sometimes be determined analytically, but is often found or refined empirically in practice. We note that the order of the spatial discretization, which affects the node spacing within the element, is often embedded within this constant.

Different time-stepping schemes may have different CFL constants, and any new scheme, especially one with local-stepping abilities, should not lower the CFL constant. Of future interest may be methods that maximize the CFL constant (and thereby  $\Delta t$ ), such as strong-stability-preserving Runge-Kutta methods [Gottlieb, 2005] or higher-order symplectic schemes [Nissen-Meyer et al., 2008].

In practice, the CFL condition in (3.14) represents a global minimum reduction on the element size (normalized by the local material velocity). In practice, this means that any small element created as a side-effect of meshing difficulties or required by small-scale features will drastically increase the number of time-steps required to achieve a particular solution.

### 3.2.2 Comparison with leapfrog

Newmark, as defined in (3.6) can be rewritten as a leapfrog scheme. We can rewrite the expressions for  $\mathbf{u}_{n+1}$  and  $\mathbf{u}_n$  in terms of  $\mathbf{v}_n$  and  $\mathbf{v}_{n-1}$

$$\mathbf{v}_n = \frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{\Delta t} - \frac{\Delta t}{2} \mathbf{a}_n, \quad (3.15)$$

$$\mathbf{v}_{n-1} = \frac{\mathbf{u}_n - \mathbf{u}_{n-1}}{\Delta t} - \frac{\Delta t}{2} \mathbf{a}_{n-1}. \quad (3.16)$$

Inserting  $\mathbf{v}_{n-1}$  from (3.16) into the expression for  $\mathbf{v}_n$  (3.6) yields

$$\begin{aligned} \mathbf{v}_n &= \mathbf{v}_{n-1} + \left( \frac{1}{2} \Delta t \mathbf{a}_{n-1} + \frac{1}{2} \Delta t \mathbf{a}_n \right) \\ &= \left( \frac{\mathbf{u}_n - \mathbf{u}_{n-1}}{\Delta t} - \frac{1}{2} \Delta t \mathbf{a}_{n-1} \right) + \left( \frac{1}{2} \Delta t \mathbf{a}_{n-1} + \frac{1}{2} \Delta t \mathbf{a}_n \right) \\ &= \frac{\mathbf{u}_n - \mathbf{u}_{n-1}}{\Delta t} + \frac{1}{2} \Delta t \mathbf{a}_n. \end{aligned} \quad (3.17)$$

Replacing  $\mathbf{v}_n$  from (3.15) and (3.17),

$$\frac{\mathbf{u}_n - \mathbf{u}_{n-1}}{\Delta t} + \frac{1}{2} \Delta t \mathbf{a}_n = \frac{\mathbf{u}_{n+1} - \mathbf{u}_n}{\Delta t} - \frac{1}{2} \Delta t \mathbf{a}_n,$$

which can be solved for  $\mathbf{u}_{n+1}$  yielding the standard leapfrog form

$$\mathbf{u}_{n+1} = 2\mathbf{u}_n - \mathbf{u}_{n-1} + \Delta t^2 \mathbf{B} \mathbf{u}_n. \quad (3.18)$$

This equivalence between leapfrog and Newmark is important because we can confirm our convergence and stability results from the existing LTS-leapfrog method.

### 3.2.3 Conservation of energy

Because conservation of energy is often an important property of a physical system, retaining this property in discrete form is desirable. For our discrete system (3.3), and no forcing ( $\mathbf{F} = 0$ ), we have the discrete energy

$$E_h = \frac{1}{2} \left( \dot{\mathbf{u}}^T \mathbf{M} \dot{\mathbf{u}} + \mathbf{u}^T \mathbf{K} \mathbf{u} \right). \quad (3.19)$$

Krenk [2006] showed that the general Newmark scheme conserves this quantity only for the parameter choices  $\gamma = \frac{1}{2}, \beta = \frac{\gamma}{2} = \frac{1}{4}$ . This choice makes the scheme implicit, which is generally not acceptable for large-scale wave propagation. Krenk does however provided a modified energy (eq. 48), that is conserved by the explicit method

$$E'_h = \frac{1}{2} \mathbf{v}^T \mathbf{M} \mathbf{v} + \frac{1}{2} \mathbf{u}^T \left( \mathbf{K} - \frac{1}{4} \Delta t^2 \mathbf{K} \mathbf{M}^{-1} \mathbf{K} \right) \mathbf{u}. \quad (3.20)$$

We can also consider the altered system

$$\frac{d^2 \mathbf{z}}{dt^2} + \underbrace{\mathbf{M}^{-\frac{1}{2}} \mathbf{K} \mathbf{M}^{-\frac{1}{2}}}_{\mathbf{A}} \mathbf{z} = 0, \quad (3.21)$$

where  $\mathbf{z} = \mathbf{M}^{\frac{1}{2}} \mathbf{u}$  and  $\mathbf{A}$  is symmetric (note that  $\mathbf{B} = -\mathbf{M}^{-1} \mathbf{K}$  is not generally symmetric). The leapfrog time-stepping scheme (3.18) conserves the following quantity for a symmetric  $\mathbf{A}$  [Diaz and Grote, 2009]:

$$E_{n+\frac{1}{2}} = \frac{1}{2} \left[ \left\langle \left( \mathbf{I} - \frac{1}{4} \Delta t^2 \mathbf{A} \right) \frac{\mathbf{z}_{n+1} - \mathbf{z}_n}{\Delta t}, \frac{\mathbf{z}_{n+1} - \mathbf{z}_n}{\Delta t} \right\rangle + \left\langle \mathbf{A} \frac{\mathbf{z}_{n+1} + \mathbf{z}_n}{2}, \frac{\mathbf{z}_{n+1} + \mathbf{z}_n}{2} \right\rangle \right], \quad (3.22)$$

where  $\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^T \mathbf{y}$  is the standard  $\mathbb{R}^N$  inner product. This energy in (3.22) is only valid when  $(\mathbf{I} - \frac{1}{4} \Delta t^2 \mathbf{A})$  is positive definite, which is also the CFL condition for the scheme (see Sec. 3.6.2). As mentioned, explicit Newmark is identical to the leapfrog scheme and therefore will also conserve this quantity for the altered system  $\mathbf{z}$ .

### 3.2.4 Fine and coarse element regions

LTS, using the framework as discussed in [Diaz and Grote, 2009; Grote and Mitkova, 2010, 2013] allows a finite-element mesh to be divided into both *fine*  $\mathbf{P}$  and *coarse*  $(\mathbf{I} - \mathbf{P})$  regions. Using these regions, we split the DOFs

$$\mathbf{u}(t) = \mathbf{P} \mathbf{u}(t) + (\mathbf{I} - \mathbf{P}) \mathbf{u}(t) = \mathbf{u}^{[\text{fine}]} + \mathbf{u}^{[\text{coarse}]}, \quad (3.23)$$

where the matrix  $\mathbf{P}$  equals 1 on the diagonal when the DOF is within the fine region, and zero elsewhere. In a continuous finite element method, such as a SEM, the wavefield  $\mathbf{u}(t)$  is defined on mesh nodes (i.e., DOFs) that are shared on element boundaries. To resolve the ambiguity of element-boundary nodes, we assume that the fine region is greedy, meaning that the coarse-fine boundary nodes belong to the fine region. Also, we note that  $(\mathbf{I} - \mathbf{P})$  will represent nodes within the coarse region.

DOFs in the fine region take smaller steps  $\Delta\tau$ , that are defined by the fine-region CFL condition. Depending on the relative size of elements in the coarse region, it uses the maximum stable integer multiple time step  $\Delta t = p\Delta\tau$ , with refinement  $p$  such that

$$p = \left\lceil \frac{\tilde{h}_{\min}^{[\text{coarse}]}}{\tilde{h}_{\min}^{[\text{fine}]}} \right\rceil \in \mathbb{N}. \quad (3.24)$$

The scheme takes  $p$  steps of size  $\Delta\tau = \frac{\Delta t}{p}$  in the fine region for every larger  $\Delta t$  step in the coarse region. If the coarse region has relatively more elements than the fine region, LTS will be able to save a large amount of computation, that can be modeled simply as

$$\text{theoretical speedup} = \frac{p \times \#[\text{fine} + \text{coarse elements}]}{p \times \#[\text{fine elements}] + \#[\text{coarse elements}]}, \quad (3.25)$$

where we note that each fine element has to do  $p$ -times more work than a coarse element to reach the final desired elapsed time.

### 3.2.5 LTS-Newmark

When approximating the integrand in (3.8), we make the following approximation,

$$\mathbf{B}((\mathbf{I} - \mathbf{P})\mathbf{u}(t) + \mathbf{P}\mathbf{u}(t)) \approx \mathbf{B}((\mathbf{I} - \mathbf{P})\mathbf{u}(t_n) + \mathbf{P}\tilde{\mathbf{u}}(\tau)),$$

where we approximate  $\mathbf{u}(t)$  as a constant over  $[t_n, t_n + \Delta t]$ , and further utilize the separate variable  $\tilde{\mathbf{u}}(\tau)$  solving the differential equation on  $\tau = [0, \Delta t]$ , which resembles our original system,

$$\begin{aligned} \frac{d\tilde{\mathbf{u}}}{d\tau}(\tau) &= \tilde{\mathbf{v}}(\tau), \\ \frac{d\tilde{\mathbf{v}}}{d\tau}(\tau) &= \mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}(t) + \mathbf{B}\mathbf{P}\tilde{\mathbf{u}}(\tau) \\ \text{with} \quad \tilde{\mathbf{v}}(0) &= 0 \text{ and } \tilde{\mathbf{u}}(0) = \mathbf{u}(t_n). \end{aligned} \quad (3.26)$$

This “fine-level” system allows us to solve the fine-region values at a finer time step that satisfies the CFL condition, and frees the coarse-level elements to take larger steps. Note that  $\mathbf{u}(t)$  is a constant within this alternate system and that the system is time reversible, that is,  $\tilde{\mathbf{u}}(\tau) = \tilde{\mathbf{u}}(-\tau)$  and  $\tilde{\mathbf{v}}(\tau) = -\tilde{\mathbf{v}}(-\tau)$ , which also sets  $\tilde{\mathbf{v}}(0) = 0$ .

We begin by solving our original system and new system in integral form:

$$\begin{aligned} \mathbf{v}(t_n + \frac{1}{2}\Delta t) &\approx \mathbf{v}(t_n - \frac{1}{2}\Delta t) + \int_{t_n - \frac{1}{2}\Delta t}^{t_n + \frac{1}{2}\Delta t} \mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}(s)|_{s=t_n} + \mathbf{B}\mathbf{P}\tilde{\mathbf{u}}(s) ds \\ &\approx \mathbf{v}(t_n - \frac{1}{2}\Delta t) + \Delta t \mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}_n + \int_{t_n - \frac{1}{2}\Delta t}^{t_n + \frac{1}{2}\Delta t} \mathbf{B}\mathbf{P}\tilde{\mathbf{u}}(s) ds, \\ \mathbf{u}(t_n + \Delta t) &\approx \mathbf{u}(t_n) + \int_{t_n}^{t_n + \Delta t} \mathbf{v}(s)|_{s=t_n + \frac{1}{2}} ds \\ &\approx \mathbf{u}(t_n) + \Delta t \mathbf{v}_{n+\frac{1}{2}}. \end{aligned} \quad (3.27)$$

Writing down the equivalent integral expressions for (3.26), we have

$$2\tilde{\mathbf{v}}(\frac{1}{2}\Delta t) = \Delta t \mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}(t) + \int_{-\frac{1}{2}\Delta t}^{\frac{1}{2}\Delta t} \mathbf{B}\mathbf{P}\tilde{\mathbf{u}}(s) ds, \quad (3.28)$$

$$\tilde{\mathbf{u}}(\Delta t) = \mathbf{u}(t_n) + \int_0^{\Delta t} \tilde{\mathbf{v}}(s) ds, \quad (3.29)$$

where we used the facts that  $\tilde{\mathbf{u}}(0) = \mathbf{u}(t_n)$  and  $\tilde{\mathbf{v}}(\frac{1}{2}\Delta t) = -\tilde{\mathbf{v}}(-\frac{1}{2}\Delta t)$ .

Writing the equations for  $\mathbf{v}(t)$  and  $\tilde{\mathbf{v}}(\tau)$  together, we have

$$\mathbf{v}(t_n + \frac{1}{2}\Delta t) - \mathbf{v}(t_n - \frac{1}{2}\Delta t) = 2\tilde{\mathbf{v}}(\frac{1}{2}\Delta t),$$

which yields a relation for the staggered step of  $\mathbf{v}_{n+\frac{1}{2}}$ ,

$$\mathbf{v}_{n+\frac{1}{2}} = 2\tilde{\mathbf{v}}(\frac{1}{2}\Delta t) + \mathbf{v}(t_n - \frac{1}{2}\Delta t). \quad (3.30)$$

This equation poses the question, how do we evaluate  $\tilde{\mathbf{v}}(\frac{1}{2}\Delta t)$ ? By approximating the expression in (3.29) using the same midpoint used for standard Newmark, we get the required expression, (3.28)

$$\begin{aligned} \tilde{\mathbf{u}}(\Delta t) &= \mathbf{u}(t_n) + \Delta t \tilde{\mathbf{v}}(\frac{1}{2}\Delta t), \\ \tilde{\mathbf{v}}(\frac{1}{2}\Delta t) &= \frac{\tilde{\mathbf{u}}(\Delta t) - \mathbf{u}(t_n)}{\Delta t}. \end{aligned} \quad (3.31)$$

Using this expression we can write the final expressions for an LTS-Newmark scheme

$$\begin{aligned}\mathbf{v}_{n+\frac{1}{2}} &= 2\frac{\tilde{\mathbf{u}}(\Delta t) - \mathbf{u}(t_n)}{\Delta t} + \mathbf{v}(t_n - \frac{1}{2}\Delta t), \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_{n+\frac{1}{2}},\end{aligned}\quad (3.32)$$

where  $\tilde{\mathbf{u}}(\Delta t)$  is the result of stepping  $\tilde{\mathbf{u}}(\tau)$  using Newmark on the fine grid for  $(p = \frac{\Delta t}{\Delta \tau})$  iterations.

With the expressions in (3.32) defining the updates on the global DOFs, we can write down the LTS-Newmark scheme for 2-levels:

**LTS-Newmark (v1) (2-level)**

Given  $\mathbf{u}_0, \mathbf{v}_{-\frac{1}{2}}, \Delta \tau = \frac{\Delta t}{p}$

Initialize  $\tilde{\mathbf{u}}_0 = \mathbf{u}_0, \tilde{\mathbf{v}}_0 = 0$

For  $n = 0 \dots T_n$

$$\mathbf{w} = \mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}_n$$

$$\tilde{\mathbf{v}}_{\frac{1}{2}} = \frac{1}{2}\Delta \tau (\mathbf{B}\mathbf{P}\mathbf{u}_n + \mathbf{w})$$

$$\tilde{\mathbf{u}}_1 = \mathbf{u}_n + \Delta \tau \tilde{\mathbf{v}}_{m+\frac{1}{2}}$$

For  $m = 1 \dots (p - 1)$

$$\tilde{\mathbf{v}}_{m+\frac{1}{2}} = \tilde{\mathbf{v}}_{m-\frac{1}{2}} + \Delta \tau \mathbf{w} + \Delta \tau \mathbf{B}\mathbf{P}\tilde{\mathbf{u}}_m$$

$$\tilde{\mathbf{u}}_{m+1} = \tilde{\mathbf{u}}_m + \Delta \tau \tilde{\mathbf{v}}_{m+\frac{1}{2}}$$

$$\mathbf{v}_{n+\frac{1}{2}} = \mathbf{v}_{n-\frac{1}{2}} + 2\left(\frac{\tilde{\mathbf{u}}_p - \mathbf{u}_n}{\Delta t}\right)$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t \mathbf{v}_{n+\frac{1}{2}}$$

### 3.3 Single-Step Method

In order to utilize the accuracy and stability results of the previously derived and analyzed LTS leapfrog scheme, it is useful to rewrite the method in a single-step formulation

$$\tilde{\mathbf{u}}_m = \mathbf{u}_n + \sum_{i=0}^{(m-1)} \alpha_i^m (\Delta \tau)^{(2i+2)} (\mathbf{B}\mathbf{P})^i \mathbf{B}\mathbf{u}_n \quad (m > 0), \quad (3.33)$$

where the constants defined by  $\alpha_i^m$  are defined by the following recurrences:

$$\begin{cases} \alpha_0^2 = 2, & \alpha_1^2 = \frac{1}{2}, \\ \alpha_0^{k+1} = \frac{(k+1)^2}{2}, \\ \alpha_i^{k+1} = 2\alpha_i^k - \alpha_i^{k-1} + \alpha_{i-1}^k, & i = 1 \dots k-2, \\ \alpha_{k-1}^{k+1} = 2\alpha_{k-1}^k + \alpha_{k-2}^k, \\ \alpha_k^{k+1} = \alpha_{k-1}^k. \end{cases} \quad (3.34)$$

In order to prove this, we must write  $\tilde{\mathbf{v}}_m$  in the following form:

$$\tilde{\mathbf{v}}_{m-\frac{1}{2}} = \sum_{i=0}^{(m-1)} \beta_i^m (\Delta\tau)^{(2i+1)} (\mathbf{BP})^i \mathbf{B}\mathbf{u}_n \quad (m > 0), \quad (3.35)$$

where we will derive solutions for  $\alpha$  and  $\beta$  using induction on  $m$ . As  $\tilde{\mathbf{v}}_m$  is purely defined in terms of  $\mathbf{u}_n$ , we will be able to eliminate it and  $\beta_i^m$ , leaving only  $\alpha_i^m$ . Starting with  $m = 2$ , for  $\tilde{\mathbf{v}}_{m-\frac{1}{2}}$  we have

$$\begin{aligned} \beta_0^2 \Delta\tau \mathbf{B}\mathbf{u}_n + \beta_1^2 \Delta\tau^3 \mathbf{BPB}\mathbf{u}_n &= \frac{1}{2} \Delta\tau \mathbf{B}\mathbf{u}_n + \Delta\tau \mathbf{BP}\tilde{\mathbf{u}}_1 + \Delta\tau \mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}_n \\ &= \frac{3}{2} \mathbf{B}\mathbf{u}_n + \frac{1}{2} \Delta\tau^3 \mathbf{BPB}\mathbf{u}_n \end{aligned}$$

leaving us with  $\beta_0^2 = \frac{3}{2}, \beta_1^2 = \frac{1}{2}$ . Repeating this for  $\tilde{\mathbf{u}}_m$  we have

$$\begin{aligned} \mathbf{u}_n + \alpha_0^2 \Delta\tau^2 \mathbf{B}\mathbf{u}_n + \alpha_1^2 \Delta\tau^4 \mathbf{BPB}\mathbf{u}_n &= \tilde{\mathbf{u}}_1 + \Delta\tau \left( \frac{3\Delta\tau}{2} \mathbf{B}\mathbf{u}_n + \frac{\Delta\tau^3}{2} \mathbf{BPB}\mathbf{u}_n \right) \\ &= \mathbf{u}_n + \frac{4\Delta\tau^2}{2} \mathbf{B}\mathbf{u}_n + \frac{\Delta\tau^4}{2} \mathbf{BPB}\mathbf{u}_n \quad (3.36) \end{aligned}$$

and verify  $\alpha_0^2 = 2$  and  $\alpha_1^2 = \frac{1}{2}$ . For the general case  $m = k$ , we begin with  $\tilde{\mathbf{v}}_{k-\frac{1}{2}}$

$$\begin{aligned} \sum_{i=0}^{k-1} \beta_i^k (\Delta\tau)^{2i+1} (\mathbf{BP})^i \mathbf{B}\mathbf{u}_n &= \sum_{i=0}^{k-2} \beta_i^{k-1} (\Delta\tau)^{2i+1} (\mathbf{BP})^i \mathbf{B}\mathbf{u}_n + \Delta\tau \mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}_n \\ &\quad + \Delta\tau \mathbf{BP} \left( \mathbf{u}_n + \sum_{i=0}^{k-2} \alpha_i^{k-1} (\Delta\tau)^{2i+2} (\mathbf{BP})^i \mathbf{B}\mathbf{u}_n \right). \end{aligned}$$

Bringing the  $\Delta\tau\mathbf{BP}$  into the second sum on the right-hand side (RHS), we can change the sum index and bound, in order to combine the first and second sum,

$$\begin{aligned} \sum_{i=0}^{k-1} \beta_i^k (\Delta\tau)^{2i+1} (\mathbf{BP})^i \mathbf{B}\mathbf{u}_n &= (\beta_0^{k-1} + 1) \Delta\tau \mathbf{B}\mathbf{u}_n \\ &+ \sum_{i=1}^{k-2} (\alpha_{i-1}^{k-1} + \beta_i^{k-1}) (\Delta\tau)^{2i+1} (\mathbf{BP})^i \mathbf{B}\mathbf{u}_n \\ &+ \alpha_{k-2}^{k-1} (\Delta\tau)^{2(k-1)+2} (\mathbf{BP})^{(k-1)} \mathbf{B}\mathbf{u}_n, \end{aligned}$$

leaving us with the formulas

$$\begin{cases} \beta_0^{k+1} = \beta_k + 1, \\ \beta_i^{k+1} = \beta_i^k + \alpha_{i-1}^k, & 0 < i < k-1, \\ \beta_k^{k+1} = \alpha_{k-1}^k. \end{cases} \quad (3.37)$$

Repeating this for  $\tilde{\mathbf{u}}_k$ ,

$$\begin{aligned} \mathbf{u}_n + \sum_{i=0}^{k-1} \alpha_i^k (\Delta\tau)^{2i+2} (\mathbf{BP})^i \mathbf{B}\mathbf{u}_n &= \mathbf{u}_n + \sum_{i=1}^{k-2} (\alpha_i^{k-1}) (\Delta\tau)^{2i+2} (\mathbf{BP})^i \mathbf{B}\mathbf{u}_n \\ &+ \Delta\tau \sum_{i=0}^{k-1} \beta_i^k (\Delta\tau)^{2i+1} (\mathbf{BP})^i \mathbf{B}\mathbf{u}_n. \end{aligned}$$

After combining the RHS sums, we have

$$\begin{aligned} \mathbf{u}_n + \sum_{i=0}^{k-1} \alpha_i^k (\Delta\tau)^{2i+2} (\mathbf{BP})^i \mathbf{B}\mathbf{u}_n &= \mathbf{u}_n + \sum_{i=1}^{k-2} (\alpha_i^{k-1} + \beta_i^k) (\Delta\tau)^{2i+2} (\mathbf{BP})^i \mathbf{B}\mathbf{u}_n \\ &+ \beta_{k-1}^k (\Delta\tau)^{2(k-1)+2} (\mathbf{BP})^{(k-1)} \mathbf{B}\mathbf{u}_n, \end{aligned}$$

yielding the formula for  $\alpha_i^k$  in terms of  $\beta_i^k$ ,

$$\begin{cases} \alpha_i^{k+1} = \alpha_i^k + \beta_i^{k+1}, & i < k-1, \\ \alpha_{k-1}^{k+1} = \beta_{k-1}^{k+1}. \end{cases} \quad (3.38)$$

From (3.37), we replace  $\beta_0^{k+1} = \beta_k + 1$  and  $\beta_i^{k+1} = \beta_i^k + \alpha_{i-1}^k$  in the above to yield

$$\begin{cases} \alpha_0^{k+1} = \alpha_0^k + \beta_0^k + 1, \\ \alpha_i^{k+1} = \alpha_i^k + \beta_i^{k+1}, & 0 < i < k-1, \\ \alpha_{k-1}^{k+1} = \beta_{k-1}^{k+1}. \end{cases}$$

From (3.38), we know that  $\beta_i^k = \alpha_i^k - \alpha_i^{k-1}$ , but only for  $i < k - 1$ . For the  $i = k - 1$  case, we are able to use  $\alpha_{k-1}^k = \beta_{k-1}^k$ . One can also quickly verify by induction that  $\beta_0^k = (k - 1) + 1/2$  leaving us with

$$\begin{cases} \alpha_0^{k+1} = \alpha_0^k + k + \frac{1}{2}, \\ \alpha_i^{k+1} = 2\alpha_i^k - \alpha_i^{k-1} + \alpha_{i-1}^k, & 0 < i < k - 1, \\ \alpha_{k-1}^{k+1} = 2\alpha_{k-1}^k + \alpha_{k-2}^k, \\ \alpha_k^{k+1} = \beta_k^{k+1}. \end{cases}$$

Finally, it is trivial to show that  $\alpha_0^{k+1} = \frac{(k+1)^2}{2}$ , proving our original assertion in (3.34).

### 3.3.1 LTS as modification to matrix B

Given a single step method for  $\tilde{\mathbf{u}}_m$ , we can write our LTS algorithm as the following:

$$\begin{aligned} \mathbf{v}_{n+\frac{1}{2}} &= \mathbf{v}_{n-\frac{1}{2}} + \Delta t \mathbf{B}_p \mathbf{u}_n, \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_{n+\frac{1}{2}}, \end{aligned}$$

where we require the following definition

$$\mathbf{B}_p = \mathbf{B} + \frac{2}{p^2} \sum_{i=1}^{p-1} \alpha_i^p (\Delta \tau)^{2i} (\mathbf{B}\mathbf{P})^i \mathbf{B}. \quad (3.39)$$

This result is best seen by rewriting  $\mathbf{v}_{n+\frac{1}{2}}$  using the new single-step result

$$\mathbf{v}_{n+\frac{1}{2}} = \mathbf{v}_{n-\frac{1}{2}} + \frac{2}{\Delta t} \left( \sum_{i=0}^{p-1} \alpha_i^p (\Delta \tau)^{2i+2} (\mathbf{B}\mathbf{P})^i \mathbf{B} \mathbf{u}_n \right).$$

After pulling out the first term in the sum and simplifying, we achieve

$$\mathbf{v}_{n+\frac{1}{2}} = \mathbf{v}_{n-\frac{1}{2}} + \Delta t \left( \mathbf{B} + \frac{2}{p^2} \sum_{i=1}^{p-1} \alpha_i^p (\Delta \tau)^{2i} (\mathbf{B}\mathbf{P})^i \mathbf{B} \right) \mathbf{u}_n,$$

demonstrating the single-step equivalence.

### 3.3.2 Equivalence to LTS-leapfrog

Given that the explicit Newmark used here is identical to a leapfrog scheme, it is not surprising that the LTS variant is also equal to LTS-leapfrog. Because of this, we are able to verify our stability and accuracy results for the 2-level scheme.

We must note however, that the LTS-leapfrog scheme solves a modified system, which causes a change of sign when computing  $\alpha_i^k$  and  $\mathbf{B}_p$ . The system uses an  $\mathbf{A}$  matrix, instead of  $\mathbf{B}$ , and has an alternate definition

$$\mathbf{A} = \mathbf{M}^{-\frac{1}{2}} \mathbf{K} \mathbf{M}^{-\frac{1}{2}}, \quad (3.40)$$

compared to  $\mathbf{B} = -\mathbf{M}^{-1} \mathbf{K}$ , which has an additional sign and is not symmetric like  $\mathbf{A}$ .

## 3.4 Absorbing Boundaries

Many applications using wave-propagation require simulations of a finite domain, creating an artificial domain boundary. Without absorbing boundaries, the simulation must be run on a very large domain, such that the waves do not reflect back to the domain of interest. Absorbing boundaries designed for acoustic and elastic wave propagation has been an active area of research, where absorbing boundary conditions (ABC) [Engquist and Majda, 1977; Clayton and Engquist, 1977; Stacey, 1988; Zampieri and Tagliani, 1997] and, perfectly matched layers (PML) [Komatitsch and Tromp, 2003] are the commonly used solutions. ABC is the simplest to implement, but only provides exact absorption when the wave's angle of incidence is normal to the domain boundary, decreasing in effectiveness as the angle increases. This approximation can be improved by increasing the order, (ABC1, ABC2, etc), however only ABC1 and ABC2 are commonly used. ABC1 is straightforward to implement, utilizing existing derivatives, where ABC2, although more accurate for off-normal reflections, requires higher-order derivatives and additional mixed derivatives that require additional computation. PML can provide near-perfect absorption, but we leave adapting LTS-Newmark to a PML-equipped solver for future work.

In general, the first-order condition (ABC1) sets an equality between a spatial derivative and a temporal one, and for the elastic case this is [Komatitsch and Tromp, 1999]:

$$\mathbf{T} \cdot \mathbf{n} = \rho [v_n(\mathbf{n} \cdot \partial_t \mathbf{u}) \mathbf{n} + v_1(\mathbf{t}_1 \cdot \partial_t \mathbf{u}) \mathbf{t}_1 + v_2(\mathbf{t}_2 \cdot \partial_t \mathbf{u}) \mathbf{t}_2],$$

where  $\mathbf{t}_1$  and  $\mathbf{t}_2$  are orthogonal unit vectors tangential to the absorbing boundary  $\Gamma$  with outward normal  $\mathbf{n}$ ,  $v_n$  is the P-wave speed in the  $\mathbf{n}$  direction,  $v_1$  is the S-wave speed in the  $\mathbf{t}_1$  direction, and  $v_2$  is the S-wave speed in the  $\mathbf{t}_2$  direction.

Rewriting the weak formulation of our original elastic system after an integration by parts, we have

$$\begin{aligned} & \int_{\Omega} \rho \mathbf{w} \cdot \partial_t^2 \mathbf{u} d\Omega \\ &= - \int_{\Omega} \nabla \mathbf{w} : \mathbf{T} d\Omega \\ & \quad + \int_{\Gamma} \rho [v_n (\mathbf{n} \cdot \partial_t \mathbf{u}) \mathbf{n} + v_1 (\mathbf{t}_1 \cdot \partial_t \mathbf{u}) \mathbf{t}_1 + v_2 (\mathbf{t}_2 \cdot \partial_t \mathbf{u}) \mathbf{t}_2] \cdot \mathbf{w} d\Gamma, \end{aligned} \quad (3.41)$$

where  $\Omega$  is our simulation domain,  $\Gamma$  is the absorbing boundary surface, and  $\mathbf{w}$  represents our test function. Following the standard discretization using GLL points and quadrature rule, we are given the following semidiscrete equation

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{C}\dot{\mathbf{u}} + \mathbf{K}\mathbf{u} = 0,$$

similar to the previously described system in (3.3), however, with the matrix  $\mathbf{C}$ , representing our absorbing boundaries, and similar to  $\mathbf{M}$ , diagonal, but only defined on domain-boundary nodes.

The addition of a term including  $\dot{\mathbf{u}}$  requires certain changes to the scheme. The standard scheme of (3.6) is rewritten as

$$\begin{aligned} \mathbf{v}_{n+\frac{1}{2}} &= \mathbf{v}_{n-\frac{1}{2}} - \Delta t \mathbf{M}^{-1} (\mathbf{K}\mathbf{u}_n + \mathbf{C}\mathbf{v}_n), \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_{n+\frac{1}{2}}, \end{aligned}$$

where  $\mathbf{v}_n$  must be computed,

$$\begin{aligned} \mathbf{v}_n &= \mathbf{v}_{n-\frac{1}{2}} - \Delta t \mathbf{M}^{-1} (\mathbf{K}\mathbf{u}_n + \mathbf{C}\mathbf{v}_n) \\ &= (\mathbf{M} + \frac{1}{2} \Delta t \mathbf{C})^{-1} (\mathbf{M}\mathbf{v}_{n-\frac{1}{2}} - \frac{1}{2} \Delta t \mathbf{K}\mathbf{u}_n), \end{aligned} \quad (3.42)$$

where we note that the computation of  $(\mathbf{M} + \frac{1}{2} \Delta t \mathbf{C})^{-1}$  is computed trivially as  $\mathbf{M}$  and  $\mathbf{C}$  are both diagonal. This formula for  $\mathbf{v}_n$  can be used directly, but with some algebra, we can rewrite the entire scheme as follows:

$$\begin{aligned} \mathbf{v}_{n+\frac{1}{2}} &= \mathbf{v}_{n-\frac{1}{2}} - \Delta t (\mathbf{M} + \frac{1}{2} \Delta t \mathbf{C})^{-1} (\mathbf{K}\mathbf{u}_n + \mathbf{C}\mathbf{v}_{n-\frac{1}{2}}), \\ \mathbf{u}_n &= \mathbf{u}_n + \Delta t \mathbf{v}_{n+\frac{1}{2}}, \end{aligned}$$

where we instead use a modified mass matrix, and the value  $\mathbf{v}_{n-\frac{1}{2}}$  on the boundary directly.

We provide below the 2-level version of the LTS-Newmark algorithm that provides absorbing boundaries, using the mass matrix correction.

**LTS-Newmark (v2)** (2-level with ABC)

$$\begin{aligned}
&\text{Given } \mathbf{u}_0, \mathbf{v}_{-\frac{1}{2}}, \Delta\tau = \frac{\Delta t}{p}, \mathbf{B} = -(\mathbf{M} + \frac{\Delta t}{2}\mathbf{C})^{-1}\mathbf{K} \\
&\text{Initialize } \tilde{\mathbf{u}}_0 = \mathbf{u}_0, \tilde{\mathbf{v}}_0 = \mathbf{0} \\
&\text{For } n = 0 \dots T_n \\
&\quad \tilde{\mathbf{v}}_{\frac{1}{2}} = \frac{1}{2}\Delta\tau\mathbf{B}\mathbf{u}_n \\
&\quad \tilde{\mathbf{u}}_1 = \mathbf{u}_n + \Delta\tau\tilde{\mathbf{v}}_{m+\frac{1}{2}} \\
&\quad \text{For } m = 1 \dots (p-1) \\
&\quad\quad \tilde{\mathbf{v}}_{m+\frac{1}{2}} = \tilde{\mathbf{v}}_{m-\frac{1}{2}} + \Delta\tau\mathbf{B}(\mathbf{I} - \mathbf{P})(\mathbf{u}_n) + \Delta\tau\mathbf{B}\mathbf{P}\tilde{\mathbf{u}}_m \\
&\quad\quad \tilde{\mathbf{u}}_{m+1} = \tilde{\mathbf{u}}_m + \Delta\tau\tilde{\mathbf{v}}_{m+\frac{1}{2}} \\
&\quad \mathbf{v}_{n+\frac{1}{2}} = \mathbf{v}_{n-\frac{1}{2}} + 2\left(\frac{\tilde{\mathbf{u}}_p - \mathbf{u}_n}{\Delta t}\right) - \Delta t(\mathbf{M} + \frac{\Delta t}{2}\mathbf{C})^{-1}\mathbf{C}\mathbf{v}_{n-\frac{1}{2}} \\
&\quad \mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t\mathbf{v}_{n+\frac{1}{2}}
\end{aligned}$$

We note that the only differences from the previous Newmark, are the modified mass matrix and the the addition of the term  $\Delta t(\mathbf{M} + \frac{\Delta t}{2}\mathbf{C})^{-1}\mathbf{C}\mathbf{v}_{n-\frac{1}{2}}$ , which is done only once at each global step, on the full domain boundary across all p-levels.

We wish to additionally highlight that in this class of LTS scheme, the fine steps  $\frac{\Delta t}{p}$  cannot be regarded as an interpolation or extrapolation of the coarse-level steps. The system  $\tilde{\mathbf{u}}(\tau)$  has a uniquely different initial condition, and because of this, we must be careful how we apply the absorbing boundary to match the non-LTS version. This may also be a factor when considering additions to the PDE model such as attenuation or damping, which might require the velocity  $\tilde{\mathbf{v}}(\tau)$  at intermediate time steps. Nonconserving LTS methods such as LTS-ABk [Grote and Mitkova, 2013] and LTS-RKK [Grote et al., 2014] are a better choice in this case and are known to work with damping.

### 3.4.1 Multiple refinement levels

For simplicity, we previously limited ourselves to 2 levels, but many applications are able to benefit significantly by adding the ability to step using an arbitrary number of levels. With only 2 levels, the mesh is divided to minimize total work needed per global step, however, this means many elements take larger-than-necessary steps. By allowing multiple levels, we add flexibility to the time steps, such that more elements are closer to their own optimal time step.

A move to multiple levels requires further variables, which we index by level

$k$  with  $k = 1 \dots k_{\max}$  from coarsest to finest; we define the following variables:

1.  $\mathbf{P}_k$  is a diagonal matrix with value 1 when the diagonal entry corresponds to level- $k$  DOFs with the following properties,

$$\sum^N \mathbf{P}_k = \mathbf{I}, \quad \mathbf{P}_j \mathbf{P}_k = 0, j \neq k. \quad (3.43)$$

2.  $\mathbf{u}_m^{(k)} = \mathbf{P}_k \mathbf{u}$ ,  $\mathbf{v}_{m/M}^{(k)} = \mathbf{P}_k \mathbf{v}$ , and  $\mathbf{a}_m^{(k)} = \mathbf{P}_k \mathbf{a}$  at step  $m$  of  $M_k$  total steps.
3.  $m$  is the step relative to the coarser neighbor, and  $M_k = \frac{dt/p_{\text{coarser-level}}}{dt/p_{\text{this-level}}}$ , i.e., the number of steps before this level is complete within a coarser level.

With two levels, we simply had  $\Delta\tau = \frac{\Delta t}{p}$ , which we extend to include more levels with time steps

$$\Delta\tau^{(k)} = \frac{\Delta t}{p(k)} \quad (3.44)$$

defined by the level  $k$  and the refinement  $p(k)$ . For this current method, we are restricted to refinements  $p(k)$  such that each successive time step is an even divisor of all previous levels,

$$\frac{p(k_2)}{p(k_1)} \in \mathbb{N}, p(k_2) \geq p(k_1).$$

With even divisors, the synchronization between levels happens at every time step (except the finest level). For instance, we are allowed to have two neighboring  $p$ -levels with time steps equal to  $\Delta t/2$  and  $\Delta t/4$ , but we do not allow neighboring  $p$ -levels with  $\Delta t/2$  and  $\Delta t/3$ . To solve this problem in an automatic way, we utilize a refinement partitioning with  $p(k) = 2^k$  yielding time-step sizes

$$\Delta\tau^{(k)} = \frac{\Delta t}{2^{k-1}} = \Delta t, \Delta t/2, \Delta t/4, \Delta t/8, \Delta t/16, \dots \text{ for } k = 1, 2, \dots \quad (3.45)$$

This guarantees that the  $p$ -levels boundaries fit together regardless of mesh structure, as it is commonly the case that  $p$ -levels share neighbors with time steps that are 2, 4, or 8 times smaller or larger (in 2D/3D examples). We additionally need to adapt the theoretical speedup model (3.25) to multiple levels,

$$\text{theoretical speedup} = \frac{p_{\max} \times \#\{\text{all elements}\}}{\sum_{i=1}^{k_{\max}} p_i \times \#\{\text{elements level } i\}}. \quad (3.46)$$

The multilevel LTS time-stepping algorithm is best understood recursively, such that each level  $k$  is embedded within its coarser level  $(k - 1)$ , up to the

coarsest level  $k = 1$ . For 3 levels, we would step with step sizes in the following order:

$$\left\{ \frac{\Delta t}{4}, \frac{\Delta t}{4}, \frac{\Delta t}{2}, \frac{\Delta t}{4}, \frac{\Delta t}{4}, \frac{\Delta t}{2}, \Delta t \right\}. \quad (3.47)$$

In order to clarify the scheme, we will briefly explain a 3-level version, mindful of the many-level generalization. Following the original 2-level derivation, we recall the ODE system for  $\tilde{\mathbf{u}}(\tau) = \mathbf{u}^{(2)}(\tau)$  and  $\tilde{\mathbf{v}}(\tau) = \mathbf{v}^{(2)}(\tau)$  on 2-levels, however, with the introduction of the third embedded variable  $\hat{\mathbf{u}}(s) = \mathbf{u}^{(3)}(s)$ ,

$$\begin{aligned} \frac{d\tilde{\mathbf{u}}}{d\tau}(\tau) &= \tilde{\mathbf{v}}(\tau), \\ \frac{d\tilde{\mathbf{v}}}{d\tau}(\tau) &= \mathbf{BP}_1 \mathbf{u}(t) + \mathbf{BP}_2 \tilde{\mathbf{u}}(\tau) + \mathbf{BP}_3 \hat{\mathbf{u}}(s). \end{aligned}$$

This third variable is governed by the following system,

$$\begin{aligned} \frac{d\hat{\mathbf{u}}}{ds}(s) &= \hat{\mathbf{v}}(s), \\ \frac{d\hat{\mathbf{v}}}{ds}(s) &= \mathbf{BP}_1 \mathbf{u}(t) + \mathbf{BP}_2 \tilde{\mathbf{u}}(\tau) + \mathbf{BP}_3 \hat{\mathbf{u}}(s), \end{aligned}$$

where  $\mathbf{u}(t)$  and  $\tilde{\mathbf{u}}(\tau)$  are constant,  $\hat{\mathbf{u}}(0) = \tilde{\mathbf{u}}(\tau)$ , and  $\hat{\mathbf{v}}(0) = 0$ , precisely as in the definition of  $\tilde{\mathbf{u}}(\tau)$ . In a recursive manner, to solve  $\mathbf{u}(t_i + \Delta t)$ , we must solve  $\tilde{\mathbf{u}}(\Delta t)$ , which requires  $i = \Delta t / \Delta \tau$  steps. However, each  $\Delta \tau$  step require us to solve  $\hat{\mathbf{u}}(\Delta \tau)$ , requiring  $j = \Delta \tau / \Delta s$  steps of  $\Delta s$ . In order to make this explicit, we write this 3-level scheme below, where we note the special care needed at the intermediate level, when  $m = 0$  and  $m > 0$ .

**LTS-Newmark (v3) (3-level)**

Given  $\mathbf{u}_0, \mathbf{v}_{-\frac{1}{2}}, \Delta\tau = \frac{\Delta t}{p_2}, \Delta s = \frac{\Delta\tau}{p_3}$

Initialize  $\tilde{\mathbf{u}}_0 = \mathbf{u}_0, \tilde{\mathbf{v}}_0 = \mathbf{0}$

For  $n = 0 \dots T_n$

$$\tilde{\mathbf{u}}_0 = \mathbf{u}_n$$

$$\mathbf{w} = \mathbf{BP}_1 \mathbf{u}_n$$

For  $m = 0 \dots (p_2 - 1)$

$$\hat{\mathbf{u}}_0 = \tilde{\mathbf{u}}_m$$

$$\mathbf{z} = \mathbf{BP}_2 \tilde{\mathbf{u}}_m$$

$$\hat{\mathbf{v}}_{\frac{1}{2}} = \frac{1}{2} \Delta s (\mathbf{w} + \mathbf{z} + \mathbf{BP}_3 \hat{\mathbf{u}}_0)$$

$$\hat{\mathbf{u}}_1 = \hat{\mathbf{u}}_0 + \Delta s \hat{\mathbf{v}}_{\frac{1}{2}}$$

For  $s = 1 \dots (p_3/p_2 - 1)$

$$\hat{\mathbf{v}}_{s+\frac{1}{2}} = \hat{\mathbf{v}}_{s-\frac{1}{2}} + \Delta s (\mathbf{w} + \mathbf{z} + \mathbf{BP}_3 \hat{\mathbf{u}}_s)$$

$$\hat{\mathbf{u}}_{s+1} = \hat{\mathbf{u}}_s + \Delta s \hat{\mathbf{v}}_{s+\frac{1}{2}}$$

$$\tilde{\mathbf{v}}_{m+\frac{1}{2}} = \begin{cases} (\hat{\mathbf{u}}_{p_3/p_2} - \tilde{\mathbf{u}}_m) / \Delta\tau, & m = 0 \\ \tilde{\mathbf{v}}_{m-\frac{1}{2}} + 2(\hat{\mathbf{u}}_{p_3/p_2} - \tilde{\mathbf{u}}_m) / \Delta\tau, & m > 0 \end{cases}$$

$$\tilde{\mathbf{u}}_{m+1} = \tilde{\mathbf{u}}_m + \Delta\tau \tilde{\mathbf{v}}_{m+\frac{1}{2}}$$

$$\mathbf{v}_{n+\frac{1}{2}} = \mathbf{v}_{n-\frac{1}{2}} + 2(\tilde{\mathbf{u}}_{p_2} - \mathbf{u}_n) / \Delta t$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t \mathbf{v}_{n+\frac{1}{2}}$$

**3.4.2 Single step equivalent for multiple levels**

In Sec. 3.3.1 we presented the modified  $\mathbf{B}_p$  as a way to prove equivalence to LTS leapfrog, where it has been used to prove accuracy and verify stability experimentally. Using these ideas, and the recursive nature of the multilevel scheme just outlined, we can build a multilevel equivalent  $\mathbf{B}_p$ , initially for 3 levels, and further generalized to an arbitrary number of levels. With this, we can prove accuracy and demonstrate stability.

Considering again the 3-level case, we note that each p-level simply uses Newmark; we can reuse the single-step formula (3.33) for the lowest level, with only slight modifications:

$$\hat{\mathbf{u}}_s = \tilde{\mathbf{u}}_m + \sum_{i=0}^{m-1} \alpha_i^m (\Delta s)^{(2i+2)} \mathbf{B}(\mathbf{P}_1 \mathbf{u}_n + (\mathbf{P}_2 + \mathbf{P}_3) \tilde{\mathbf{u}}_m). \quad (3.48)$$

This single-step formula allows us to define a  $\mathbf{B}$  modification that completes the work done by level 3 for use by level 2:

$$\mathbf{B}_{p_3} = \mathbf{B} + \frac{2}{(p_3/p_2)^2} \sum_{i=1}^{p_3/p_2-1} \alpha_i^{p_3/p_2} (\Delta s)^{2i} (\mathbf{B}\mathbf{P}_3)^i \mathbf{B}. \quad (3.49)$$

With this matrix, we can write a single-step version of level 2,

$$\tilde{\mathbf{u}}_m = \mathbf{u}_n + \sum_{i=0}^{m-1} \alpha_i^m (\Delta \tau)^{(2i+2)} \mathbf{B}_{p_3} (\mathbf{P}_2 + \mathbf{P}_3) \mathbf{B}_{p_3} \mathbf{u}_n. \quad (3.50)$$

This allows us to write the  $\mathbf{B}$  equivalent for level 2, which is used directly by the coarsest level,

$$\mathbf{B}_{p_2} = \mathbf{B}_{p_3} + \frac{2}{(p_2)^2} \sum_{i=1}^{p_2-1} \alpha_i^{p_3/p_2} (\Delta \tau)^{2i} (\mathbf{B}_{p_3} (\mathbf{P}_2 + \mathbf{P}_3))^i \mathbf{B}_{p_3}. \quad (3.51)$$

Thus, the 3-level LTS-Newmark scheme can be written as

$$\begin{aligned} \mathbf{v}_{n+1/2} &= \mathbf{v}_{n-1/2} + \Delta t \mathbf{B}_{p_2}, \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_{n+1/2}. \end{aligned} \quad (3.52)$$

From this 3-level scheme, it is relatively simple to write down the general scheme, where each higher level depends on the level below it, until the finest level, which terminates the recursive sequence:

$$\begin{aligned} \text{let } T_k &= p_k/p_{k-1}, \\ \mathbf{B}_{k < N} &= \mathbf{B}_{k+1} + \frac{2}{T_k} \sum_{i=1}^{(T_k-1)} \alpha_i^{T_k} (\Delta t/p_k)^{(2i)} (\mathbf{B}_{k+1} \sum_{s=k+1}^N \mathbf{P}_s)^i \mathbf{B}_{k+1}, \\ \mathbf{B}_N &= \mathbf{B} + \frac{2}{T_N} \sum_{i=1}^{(T_N-1)} \alpha_i^{T_N} (\Delta t/p_N)^{(2i)} (\mathbf{B}\mathbf{P}_N)^i \mathbf{B}. \end{aligned} \quad (3.53)$$

Thus, we can write the multilevel equivalent formulation

$$\begin{aligned} \mathbf{v}_{n+1/2} &= \mathbf{v}_{n-1/2} + \Delta t \mathbf{B}_1 \mathbf{u}_n \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_{n+1/2}, \end{aligned} \quad (3.54)$$

where  $\mathbf{B}_1$  is defined by (3.53). This scheme is second-order accurate in  $\mathbf{u}(t_n)$ . This result requires that

$$\mathbf{v}_{n+1/2} = \mathbf{v}_{n-1/2} + \Delta t \mathbf{B}_1 \mathbf{u}_n + O(\Delta t), \quad (3.55)$$

which is best proved by induction. First we note that

$$\mathbf{B}_1 \mathbf{u}_n = \mathbf{B}_2 \mathbf{u}_n + O(\Delta t^2)$$

and also that

$$\mathbf{B}_k \mathbf{u}_n = \mathbf{B}_{k+1} \mathbf{u}_n + O(\Delta t^2),$$

such that we can write

$$\mathbf{B}_1 \mathbf{u}_n = \mathbf{B} \mathbf{u}_n + O(\Delta t^2)$$

by induction and the definition of  $\mathbf{B}_N$ , which terminates the induction sequence and proves our accuracy statement in (3.55).

### 3.5 LTS-Newmark for Continuous Elements

The principle focus of this chapter is the derivation and implementation of a high-performance LTS-Newmark method. The previously defined algorithms do not make explicit considerations for an efficient implementation with continuous finite elements that can achieve the speedup predicted by (3.25). The algorithm does explicitly provide the computation of  $\mathbf{w} = \mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}_n$ , which ensures that this expensive operation is only done once per global time step. However it is unclear how to avoid the vector operations done on the fine level that are associated with nodes in  $(\mathbf{I} - \mathbf{P})$ . To better understand this, we list the vector additions of LTS-Newmark and the actual and ideal computational cost in terms of the coarse and fine DOFs. If we let  $K_{\text{coarse}}$  and  $K_{\text{fine}}$  represent the number of coarse and fine elements, one can write the computational complexity of the operations from Alg. (v1) in terms of their actual cost and the ideal cost:

Id	Operation	Actual Cost	Ideal Cost
1	$\mathbf{w} = \mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}_n$	$O(K_{\text{coarse}})$	$O(K_{\text{coarse}})$
2	$\tilde{\mathbf{a}} = \mathbf{B}\mathbf{P}\tilde{\mathbf{u}}_m$	$O(K_{\text{fine}})$	$O(K_{\text{fine}})$
3	$\tilde{\mathbf{v}}_{m+\frac{1}{2}} = \tilde{\mathbf{v}}_{m-\frac{1}{2}} + \Delta\tau\mathbf{w} + \Delta\tau\tilde{\mathbf{a}}$	$O(K_{\text{coarse}} + K_{\text{fine}})$	$O(K_{\text{fine}})$
4	$\tilde{\mathbf{u}}_{m+1} = \tilde{\mathbf{u}}_m + \Delta\tau\tilde{\mathbf{v}}_{m+\frac{1}{2}}$	$O(K_{\text{coarse}} + K_{\text{fine}})$	$O(K_{\text{fine}})$
5	$2 \left( \frac{\tilde{\mathbf{u}}_p - \mathbf{u}_n}{\Delta t} \right)$	$O(K_{\text{coarse}} + K_{\text{fine}})$	$O(K_{\text{fine}})$

In the table, the  $\mathbf{B}\mathbf{u}$  operations (1+2) on the coarse and fine region already compute the minimal set of operations necessary, assuming we can implement  $\mathbf{P}$  and  $(\mathbf{I} - \mathbf{P})$  efficiently. However, the vector operations 3, 4, and 5 are done on the full set of nodes. In order to achieve very high efficiency, we have to extract the minimal set of nodes in  $\mathbf{P}$  and  $(\mathbf{I} - \mathbf{P})$  required to initialize, execute,

and finalize the fine-region steps. If we consider a fixed fine region and growing coarse region, we want the cost of computing the fine-region steps (3,4, and 5) to remain  $O(K_{\text{fine}})$ , as the coarse region grows pushing the theoretical speedup asymptotically to  $p$ . In order to write this algorithm for continuous elements, we need to reconsider the boundaries between coarse and fine.

Most implementations never explicitly assemble  $\mathbf{K}$ , but instead provide the action of  $\mathbf{K}\mathbf{u}$  as a loop over elements. Thus, it is important to know how to handle the boundaries between  $p$ -levels, which act to mix contributions across time-stepping regions due to the the continuous nature of the finite-element basis functions. In order to characterize this mixing in the discretized system (3.4), we define two further selection matrices:

1.  $\mathbf{R} \subset (\mathbf{I} - \mathbf{P})$  — Diagonal matrix with 1 at *coarse* nodes that are in an element also containing fine nodes with properties

$$\mathbf{PBR} \neq \emptyset, \quad (3.56)$$

$$\mathbf{PB}(\mathbf{I} - \mathbf{P} - \mathbf{R}) = \emptyset. \quad (3.57)$$

In other words,  $\mathbf{R}$  selects coarse nodes that, through  $\mathbf{B}$ , contribute to the fine region.

2.  $\mathbf{F} \subset \mathbf{P}$  — Diagonal matrix with 1 at *fine* nodes that are directly bordering coarse nodes with properties

$$(\mathbf{I} - \mathbf{P})\mathbf{BF} \neq \emptyset, \quad (3.58)$$

$$(\mathbf{I} - \mathbf{P})\mathbf{B}(\mathbf{P} - \mathbf{F}) = \emptyset. \quad (3.59)$$

In other words,  $\mathbf{F}$  selects nodes that, through  $\mathbf{B}$ , contribute to the coarse region. These additional selection matrices are illustrated in Fig. 3.1 with the continuous nodal basis functions that define the DOFs.

Intuitively, the  $\mathbf{K}$  spatial operator smears values at nodes 1 and 2 into the neighboring fine element via node 3, and vice versa. In order to implement LTS-Newmark for a SEM efficiently, the use of the  $\mathbf{P}$ ,  $\mathbf{R}$ , and  $\mathbf{F}$  selection matrices is needed so that the implementation only computes the substep  $\Delta\tau$  values of coarse nodes where the fine region requires their value.

As noted, these matrices help define the flow of information across element boundaries, a crucial element of LTS. One may quickly realize that alternate finite-element formulations such as DG [Hesthaven and Warburton, 2008] can directly implement LTS without calculating these explicitly local “communication” matrices  $\mathbf{R}$  and  $\mathbf{F}$ , easing high performance LTS implementations, like those

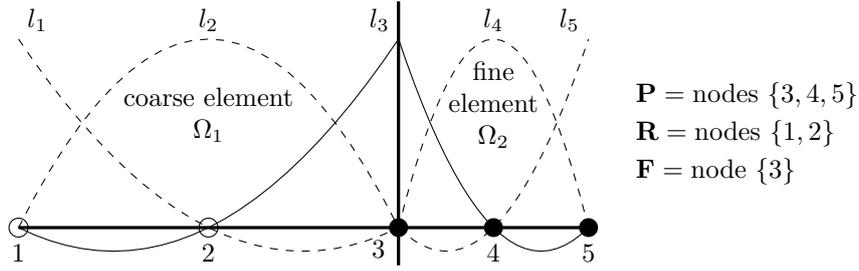


Figure 3.1. A 1D example of the interface between two  $p$ -levels from the perspective of a fine element. The  $l_i(x)$  represent a degree-two polynomial finite-element basis set on the GLL points (for  $N = 2$ ).  $l_3(x)$ , highlighted with a solid line, has support over both elements, where the  $l_{i \neq 3}$ , marked with dotted lines, only have support in their respective elements.

by Dumbser et al. [2007] and Godel et al. [2010]. The element boundaries and therefore the  $p$ -level boundaries are explicitly coupled via the numerical flux and, because of this, implementing LTS from the global formulation can be relatively straightforward. As noted in [Grote and Mitkova, 2013], the choice of spatial discretization does not impact the convergence or stability properties of the LTS method. Of course, most implementations initially choose to implement an SEM or a DG discretization for a specific reason that is beyond the scope of this chapter.

Using this perspective, we examine the terms  $\mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}_n$  and  $\mathbf{B}\mathbf{P}\tilde{\mathbf{u}}_m$  to alter their structure to utilize  $\mathbf{R}$  and  $\mathbf{F}$  to make the coupling between coarse and fine explicit. The displacement and velocity updates  $\tilde{\mathbf{u}}_{m+1}$ ,  $\mathbf{u}_{n+1}$ , and  $\mathbf{v}_{n+\frac{1}{2}}$ , are purely vectorized updates, meaning that there is no explicit coupling between spatial DOF.

A closer look at  $\tilde{\mathbf{v}}_{m+\frac{1}{2}}$  from the LTS-Newmark algorithm yields the following coarse region term

$$\tilde{\mathbf{v}}_{m+\frac{1}{2}} = \cdots + \mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}_n + \cdots,$$

which, using properties (3.56) and (3.57), can be rewritten using  $\mathbf{R}$  as

$$\tilde{\mathbf{v}}_{m+\frac{1}{2}} = \cdots + \mathbf{P}\mathbf{R}\mathbf{u}_n + (\mathbf{I} - \mathbf{P})\mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}_n + \cdots.$$

This explicitly gives us the contribution of the coarse nodes to fine nodes ( $\mathbf{P}\mathbf{R}\mathbf{u}_n$ ) and the coarse node intermediate step ( $(\mathbf{I} - \mathbf{P})\mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}_n$ ), which only contributes to coarse nodes. We can also do the inverse of this to determine the fine-node contribution to the coarse nodes using the fine-node term

$$\tilde{\mathbf{v}}_{m+\frac{1}{2}} = \cdots + \mathbf{B}\mathbf{P}\tilde{\mathbf{u}}_m + \cdots,$$

which becomes using **F** properties (3.58) and (3.59),

$$\tilde{\mathbf{v}}_{m+\frac{1}{2}} = \cdots + \mathbf{PBP}\tilde{\mathbf{u}}_m + (\mathbf{I} - \mathbf{P})\mathbf{BF}\tilde{\mathbf{u}}_m + \cdots.$$

Similarly, we see the fine-node contribution to other fine nodes in  $(\mathbf{PBP}\tilde{\mathbf{u}}_m)$  and the fine-node contribution to the coarse nodes  $((\mathbf{I} - \mathbf{P})\mathbf{BF}\tilde{\mathbf{u}}_m)$ . Because coarse-for-coarse  $(\mathbf{I} - \mathbf{P})\mathbf{B}(\mathbf{I} - \mathbf{P})$  contributions are simply additive, they can be removed from the fine region and simply added to the final, coarse-region update.

We can now rewrite the original two-level LTS-Newmark scheme that can be readily used to implement LTS-Newmark in a practical solver. This requires the reduced equality operator  $\left(\stackrel{\mathbf{Q}}{=}\right)$  that only operates on the set of nodes **Q**, where **Q** represents a selection matrix such as **P**.

**LTS-Newmark (v4) (2-level with **R** and **F**)**

$$\text{Given } \mathbf{u}_0, \mathbf{v}_{-\frac{1}{2}}, \Delta\tau = \frac{\Delta t}{p}$$

$$\text{Initialize } \tilde{\mathbf{u}}_0 = \mathbf{u}_0, \tilde{\mathbf{v}}_0 = \mathbf{0}$$

$$\text{For } n = 0 \dots T_n$$

$$\mathbf{w} \stackrel{\mathbf{P}}{=} \mathbf{PBRu}_n$$

$$\tilde{\mathbf{v}}_{\frac{1}{2}} \stackrel{\mathbf{P}+\mathbf{R}}{=} \frac{1}{2}\Delta\tau (\mathbf{PBRu}_n + \mathbf{PBP}\tilde{\mathbf{u}}_0 + (\mathbf{I} - \mathbf{P})\mathbf{BF}\tilde{\mathbf{u}}_0)$$

$$\tilde{\mathbf{u}}_1 \stackrel{\mathbf{P}+\mathbf{R}}{=} \mathbf{u}_n + \Delta\tau\tilde{\mathbf{v}}_{m+\frac{1}{2}}$$

$$\text{For } m = 1 \dots (p - 1)$$

$$\tilde{\mathbf{a}}_m \stackrel{\mathbf{P}+\mathbf{R}}{=} \mathbf{w} + \mathbf{PBP}\tilde{\mathbf{u}}_m + (\mathbf{I} - \mathbf{P})\mathbf{BF}\tilde{\mathbf{u}}_m$$

$$\tilde{\mathbf{v}}_{m+\frac{1}{2}} \stackrel{\mathbf{P}+\mathbf{R}}{=} \tilde{\mathbf{v}}_{m-\frac{1}{2}} + \Delta\tau\tilde{\mathbf{a}}_m$$

$$\tilde{\mathbf{u}}_{m+1} \stackrel{\mathbf{P}+\mathbf{R}}{=} \tilde{\mathbf{u}}_m + \Delta\tau\tilde{\mathbf{v}}_{m+\frac{1}{2}}$$

$$\mathbf{z} \stackrel{\mathbf{P}+\mathbf{R}}{=} 2 \left( \frac{\tilde{\mathbf{u}}_p - \mathbf{u}_n}{\Delta t} \right)$$

$$\mathbf{v}_{n+\frac{1}{2}} = \mathbf{v}_{n-\frac{1}{2}} + \mathbf{z} + \Delta t(\mathbf{I} - \mathbf{P})\mathbf{B}(\mathbf{I} - \mathbf{P})\mathbf{u}_n$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t \mathbf{v}_{n+\frac{1}{2}}$$

### 3.5.1 Implementation detail

This thesis is written in the context of an efficient LTS implementation that can be used to achieve the speedup LTS promises. As such, we provide pseudo code that can be used to adapt an existing code, or guide the creation of a new code.

In terms of memory usage, the LTS-Newmark and leapfrog schemes require, additionally,

- $\mathbf{u}$  and  $\mathbf{v}$  covering both the DOFs within each  $\mathbf{P}_i$ , and the DOFs for all finer levels  $P_{k>i}$ .
- Array of p-level for each element and node if the elements and DOFs cannot be rearranged according to level.
- List of elements on p-level boundaries.

In practice, one can exchange memory usage for implementation simplicity, and by grouping  $\mathbf{u}$  by level, and possibly by boundary, can have a significant effect on performance.

The multilevel LTS algorithm is recursive in nature, and is suited to a recursive implementation, which we demonstrate with the pseudocode in Alg. 6. The code uses the “ilevel” variable, which indexes the current p-level and counts from the  $p = p_{\max}$  as ilevel= 1, and  $p = 1$  as ilevel=num\_p\_levels, following the conventions in our implementation of LTS in SPECSEM3D.

## 3.6 Numerical Experiments

In order to validate the numerical accuracy and efficiency of the LTS-Newmark scheme outlined in this thesis, we have performed convergence and stability experiments of the method in one dimension using a simpler acoustic wave equation.

### 3.6.1 Numerical convergence

For initial testing, we implemented the SEM in one dimension for the acoustic wave equation with Dirichlet boundary conditions and a normalized background velocity (i.e.,  $c(x) = 1$ ):

$$\begin{aligned} \frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} &= 0 \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{3.60}$$

We ran the experiments on a domain of  $\Omega = [0, 6]$ , with only a “coarse” level  $\Omega_c = [0, 2) \cup (4, 6]$  and a “fine” level  $\Omega_f = [2, 4]$  with a refinement level  $p$  such that the fine-level time step would be  $\frac{\Delta t}{p}$  relative to the global time step  $\Delta t$ . As

---

**Algorithm 6** LTS Code Structure
 

---

```

function TIME_STEPPING
  for it= 1 ... T do
    TAKE_STEP(num_p_level)
  end for
end function
function TAKE_STEP(ilevel)
  for m = 1 ... M(ilevel) do
    if ilevel > 1 then
      TAKE_STEP(ilevel-1)
    end if
     $\mathbf{a} \leftarrow \mathbf{K}(\mathbf{P} + \mathbf{R})\mathbf{u}(\text{ilevel})$ 
    source_term( $\mathbf{a}$ ,ilevel)
    if ilevel=num_p_levels then ABS_BOUNDARY( $\mathbf{a}$ , $\mathbf{v}(\text{ilevel})$ )
    end if
    if m = 1 and ilevel < num_p_level then
       $\mathbf{v}(\text{ilevel}) \leftarrow \mathbf{v}(\text{ilevel}) + \frac{1}{2}\Delta\tau_{\text{ilevel}} \mathbf{a}$ 
      if ilevel > 1 then
         $\mathbf{v}(\text{ilevel}) \leftarrow \mathbf{v}(\text{ilevel}) + \frac{1}{\Delta\tau_{\text{ilevel}}} (\mathbf{u}(\text{ilevel-1}) - \mathbf{u}(\text{ilevel}))$ 
      end if
    else
       $\mathbf{v}(\text{ilevel}) \leftarrow \mathbf{v}(\text{ilevel}) + \Delta\tau_{\text{ilevel}} \mathbf{a}$ 
      if ilevel > 1 then
         $\mathbf{v}(\text{ilevel}) \leftarrow \mathbf{v}(\text{ilevel}) + \frac{2}{\Delta\tau_{\text{ilevel}}} (\mathbf{u}(\text{ilevel-1}) - \mathbf{u}(\text{ilevel}))$ 
      end if
    end if
     $\mathbf{u}(\text{ilevel}) \leftarrow \mathbf{u}(\text{ilevel}) + \Delta\tau_{(\text{ilevel})}\mathbf{v}(\text{ilevel})$ 
  end for
end function

```

---

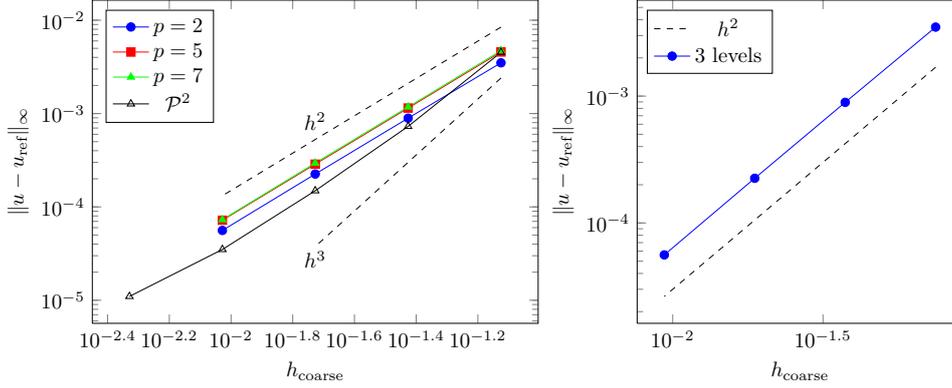


Figure 3.2. Numerical convergence for  $\mathcal{P}^1$  and  $\mathcal{P}^2$  elements using two levels (left) and  $\mathcal{P}^1$  elements on three levels (right). Tests run for  $T = 10$ .

a reference we used the exact solution

$$u_{\text{ref}}(x, t) = \sin(\pi x) \cos(\pi t),$$

and compute the error in the  $L_{\infty}$  norm. Because Newmark is  $O(\Delta t^2)$  accurate, we are limited to  $O(h^2)$  convergence using linear  $\mathcal{P}^1$  finite elements in the  $L_2$  or  $L_{\infty}$  norms. We demonstrate optimal convergence in Fig. 3.2 for varying amounts of local refinement, and also for an example with 3 levels of refinement. Some solvers choose to use  $\mathcal{P}^2$  and higher elements, despite the loss of optimal convergence  $O(h^3)$  as seen in the left panel. In some examples, increasing the spatial order of accuracy can yield a worthwhile increase in total accuracy for comparable cost.

### 3.6.2 Stability and CFL invariance

The success of our or any LTS method relies on the conditional stability of the original explicit time-stepping scheme. Given that LTS is purely designed to reduce the total work, and thus increase performance, it is critical that any LTS scheme demonstrate CFL invariance, i.e., it should share the stability properties of the non-LTS version. If the CFL condition is reduced, LTS cannot achieve the full performance predicted by the theoretical speedup (3.46).

A detailed stability analysis is critical to the success of any time-stepping scheme. In fact, several of our previous attempts to derive LTS for Newmark appeared to work successfully. However, stability analysis showed that they were in fact weakly unstable, only causing simulation “blow up” after 10,000–20,000 steps. We conjecture that breaking Newmark’s conservative nature was at the

root of this instability, but it was a hard-won lesson in the importance of careful analysis.

We do not have a mathematical proof that our multilevel Newmark scheme is (conditionally) stable; however we can run numerical stability experiments to convincingly demonstrate the stability. A key result from this (and previous leapfrog analysis) is that p-levels must utilize overlap in order to maintain stability for the largest time step as defined by the non-LTS scheme. Surprisingly, this overlap requirement goes away for Adams–Bashforth [Grote and Mitkova, 2013] and Runge–Kutta [Grote et al., 2014]. We conjecture this is related to conserving the discrete energy, and thus other energy-conserving schemes may also require overlap to maintain the full stability region.

The foundations of this stability analysis are the eigenvalues of the scheme’s update matrix. In Secs. 3.3.1 and 3.4.2 we saw that we can write our LTS–Newmark scheme for two- and multilevel schemes as a perturbation to  $\mathbf{B}$  in the original Newmark scheme

$$\begin{aligned}\mathbf{v}_{n+1} &= \mathbf{v}_n + \Delta t \mathbf{B}_p \mathbf{u}_n, \\ \mathbf{u}_{n+1} &= \mathbf{u}_n + \Delta t \mathbf{v}_{n+1},\end{aligned}$$

where  $\mathbf{B}_p$  represents the collective LTS modifications to the original spatial discretization operator  $\mathbf{B}$  (containing substeps at every level). Rewriting this scheme as a system

$$\mathbf{L} \mathbf{q}_{n+1} = \mathbf{R} \mathbf{q}_n, \quad \mathbf{q}_n = \begin{pmatrix} \mathbf{u} \\ \mathbf{v} \end{pmatrix}_n, \quad (3.61)$$

$$\mathbf{L} = \begin{pmatrix} \mathbf{I} & -\Delta t \mathbf{I} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}, \quad (3.62)$$

$$\mathbf{R} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \Delta t \mathbf{B}_p & \mathbf{I} \end{pmatrix}. \quad (3.63)$$

By moving  $\mathbf{L}$  to the RHS and computing  $\mathbf{L}^{-1} \mathbf{R}$ , we yield the following expression:

$$\mathbf{q}_{n+1} = \begin{pmatrix} \mathbf{I} + \Delta t^2 \mathbf{B}_p & \Delta t \mathbf{I} \\ \Delta t \mathbf{B}_p & \mathbf{I} \end{pmatrix} \mathbf{q}_n = \mathbf{A} \mathbf{q}_n, \quad (3.64)$$

where  $\mathbf{A}$  now contains the necessary stability information via its largest eigenvalues. These eigenvalues are a function of  $\Delta t$  and, for stability, the system should satisfy

$$\|\lambda(\mathbf{A})\| \leq 1$$

for all eigenvalues.

Using the same 1D experimental setup (3.60) from Sec. 3.6.1, with a  $p$ -refined center region, we build a one-step update matrix  $\mathbf{A}$ . For this particular setup, the CFL constant for standard explicit Newmark is simply one, meaning that the maximum stable time step is

$$\Delta t_{\text{opt}} = h_{\text{min}}. \quad (3.65)$$

Thus for a setup with a center region with twice the element density ( $p = 2$ ), we want to have  $\Delta t = h_{\text{coarse}}$  and  $\Delta \tau = h_{\text{coarse}}/p = h_{\text{coarse}}/2$ . However, as discussed, we are required to introduce *overlap* between the coarse and fine regions, that is the fine region needs to expand by 1 or more elements. We present the maximum (magnitude) eigenvalues of  $\mathbf{A}$  in Fig. 3.3. It is clear to see that the mesh without

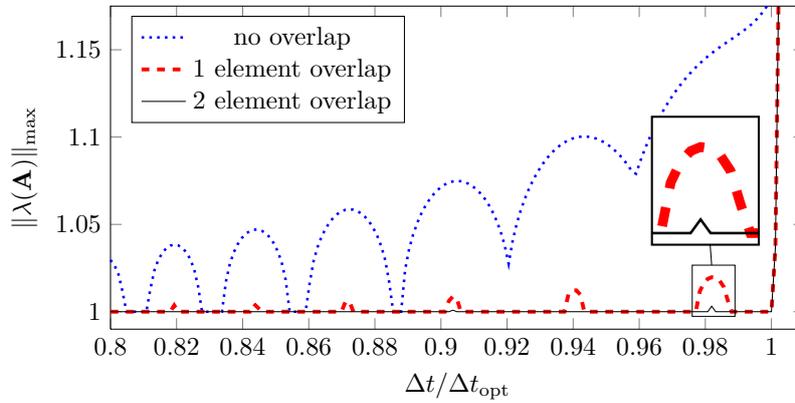


Figure 3.3. Eigenvalue behavior for mesh with center refinement of  $p = 2$  showing no overlap, overlap by 1, overlap by 2. Note that both overlap cases are stable for  $\Delta t = \Delta t_{\text{opt}}$  despite the brief oscillations with  $\|\lambda(\mathbf{A})\|_{\text{max}} > 1$ .

overlap is not stable for the optimal time step  $\Delta t_{\text{opt}}$  and that this stability is corrected for the overlap by 1- and 2-element cases. However, we do notice the peculiar oscillatory behavior that remains (hardly visible for overlap by 2). Despite these small regions of instability, the LTS scheme is able to take full-size time steps that match the standard Newmark in size for the coarse region, meaning that we have not lost any time-stepping efficiency by introducing LTS.

The 1D case presents the particular situation that the transition from large to small elements can be arbitrarily large. In 2D and 3D this transition happens more slowly as the mesh cannot transition to a small element arbitrarily fast while maintaining elements of sufficient quality. This slower transition could eliminate the overlap requirement in practice, however the architecture of our

3D version does not allow us to easily test this conjecture in more than one dimension using eigenvalue checks — only ad hoc stability checks are possible.

In order to explore this effect in one dimension, we created a sequence of meshes that transition more slowly from the coarse to fine elements. Depicted in Fig. 3.4, we show how many elements are required to have stability without overlap at the optimum  $\Delta t_{\text{opt}}$  (transition length 28). Stability, for this particular case, requires a relatively slow transition, especially considering that an overlap of 1 has a similar effect on stability. From this 1D example, we can conjecture that overlap will be likely necessary in 2D and 3D due to the long necessary transition (for stability).

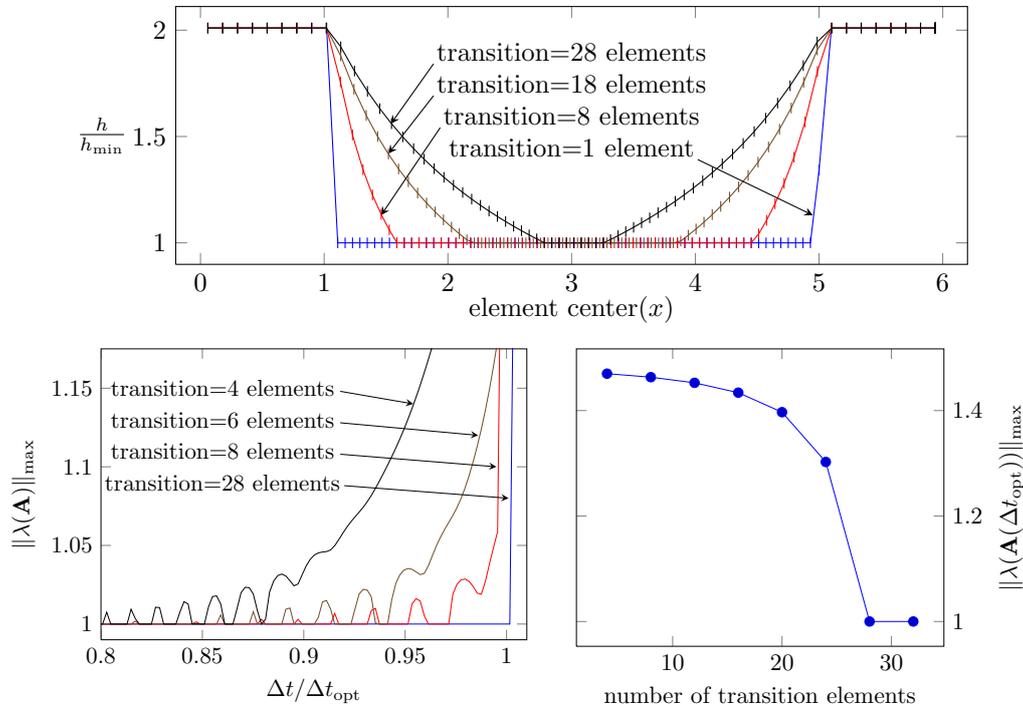


Figure 3.4. How the coarse-to-fine transition effects the stability of the LTS scheme. (top) A sequence of 1D meshes, with different transition lengths (1–28). The y-axis depicts the size of each element relative to the fine level. (bottom left) Maximum eigenvalue as a function of  $\Delta t$  for a selection of transition meshes. (bottom right) The maximum eigenvalue at  $\Delta t = \Delta t_{\text{opt}}$  for a sequence of meshes with a growing number of transition elements.

## 3.7 Implementation in Three Dimensions

As one can imagine, implementing LTS, or porting an existing code to LTS, can represent a significant developer investment, with a fairly large amount of “implementation detail” over the original Newmark scheme. This section will give an overview of our implementation of the LTS-Newmark (v4) algorithm in a production 3D wave-propagation code, as well as challenges associated with parallelization and scaling on multinode clusters.

### 3.7.1 Implementation in SPECSEM3D

The previously shown experiments were simple 1D MATLAB implementations, where we can store the full discretization operator  $\mathbf{B}$ , and can use  $\mathbf{P}$  and  $(\mathbf{I} - \mathbf{P})$  to select coarse and fine nodes, ignoring any efficiency concerns.

In order to test the method in three dimensions, we implemented LTS-Newmark in the package SPECSEM3D Cartesian [Peter et al., 2011; Komatitsch and Tromp, 2002], which was extensively introduced in Ch. 2. Recall that SPECSEM3D is a comprehensive Fortran (and now CUDA) code implementing both the viscoelastic and acoustic wave equation on large heterogeneous domains, with a focus on regional seismology. As a SEM, it uses hexahedral elements, usually with  $\mathcal{P}^4$  elements, and can automatically recommend a time step based on previous CFL experiments, which is chosen by the user at runtime.

SPECSEM3D is a robust, production ready package for both forward and inverse modeling of both acoustic and viscoelastic wave propagation. SPECSEM3D’s use of hexahedral elements enables an efficient distribution of nodal DOFs for large-scale wave-propagation problems. Hexahedral elements are also required to construct a diagonal mass matrix. The standard set of GLL quadrature points produce an unstable quadrature rule for the mass matrix on triangular and tetrahedral elements. In order to use these GLL points for 3D problems, we are required to use hexahedral elements.

In contrast to tetrahedral elements, there is no elegant algorithm to construct unstructured conforming meshes using hexahedral elements. The meshing software Trellis (née CUBIT) used to produce user-defined hexahedral meshes for SPECSEM3D, uses many heuristics and hand-tuned algorithms in order to generate meshes of reasonable quality. Any spatially heterogeneous mesh, where any small-scale features or topography (either external or internal) are required, can lead to pinching or squeezed elements that are much smaller than the average element. The contribution of the LTS-Newmark scheme to SPECSEM3D is twofold: first we reduce bottlenecks in existing applications where small el-

elements could not be eliminated. Second, we enable applications where small elements are required but were deemed too expensive due to these same CFL bottlenecks.

### 3.7.2 Experiments in three dimensions

Our initial experiments were done using a test mesh with a homogeneous velocity model and a refinement region at its center as pictured in Fig. 3.5. Given the

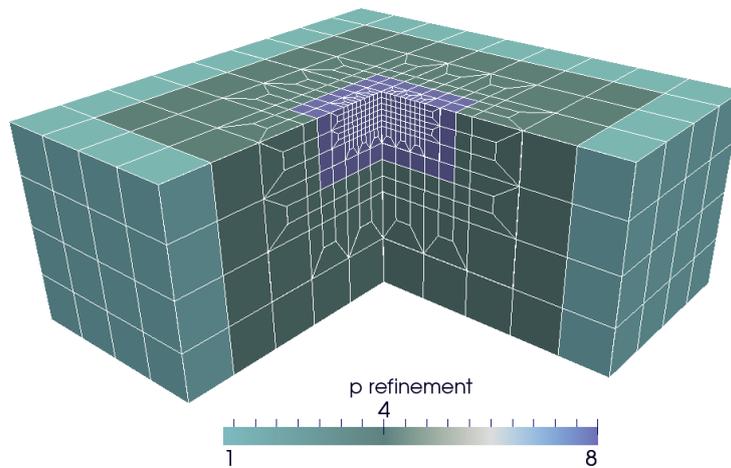


Figure 3.5. Cutaway of a mesh with three local refinement levels. The smallest elements in the middle of the model require an 8x smaller time step size than the coarsest elements located at the boundary.

seismology focus of this project, we focused on testing the implementation using localized earthquake sources (usually within a single element) and measuring the resulting solutions at or near the surface at localized stations, modeling a real-world simulation scenario.

### 3.7.3 LTS evaluation and validation

In order to test our LTS implementation in SPEC3D, we designed the mesh seen in Fig. 3.5, with coarse boundaries and refinement in the center. The test setup is depicted in Fig. 3.6, with a model earthquake near the center of the mesh at 25-km depth, and two linear arrays of stations to record solutions at the surface.

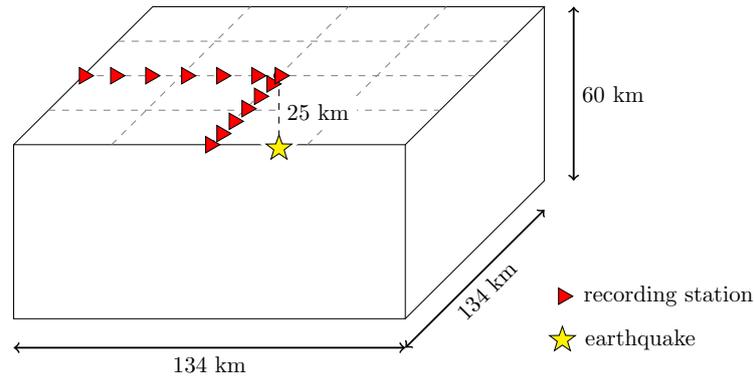


Figure 3.6. Mesh setup with 13 surface recording points and an earthquake at 25-km depth.

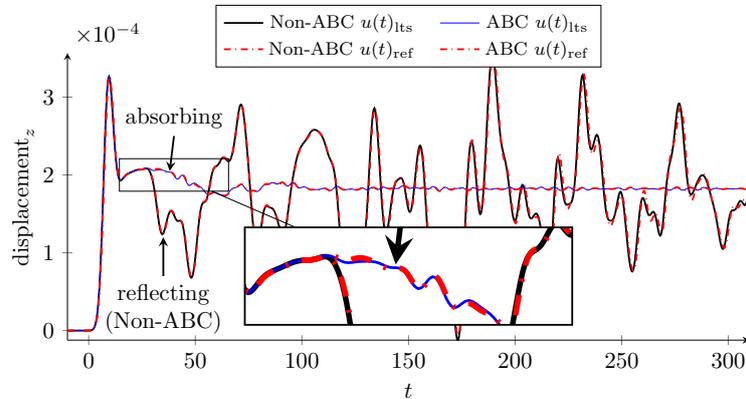


Figure 3.7. Seismogram comparison between LTS and non-LTS for both absorbing and nonabsorbing boundaries. The zoom shows that LTS and reference solutions match well for both types of boundary conditions.

Figure 3.7 depicts a recording of vertical ( $z$  – dir) displacement of a centrally located station, with absorbing (ABC) and reflecting (non-ABC) boundaries comparing the reference solution to our new LTS-Newmark scheme. Both seismogram recordings match very well for both reflecting and absorbing boundary conditions. This test example used a small mesh and the implementation can achieve almost 100% of the small predicted speedup of 1.3x.

### 3.7.4 LTS scaling

The mesh used to validate the implementation had a very modest theoretical speedup of only 1.3x. To ensure that the desired overhead complexities from

the table in Sec. 3.5 are correctly implemented, we need to test this on a mesh with much higher speedup potential. We created a series of meshes with approximately 300,000 elements (19M DOFs with 125 nodes per element) with theoretical speedups (3.46) ranging from 2x–100x relative to the non-LTS version of the code. These meshes use an iteratively refined zone of elements at the surface, similar to our verification mesh in Fig. 3.5, except with a refinement between  $p = 2$  and  $p = 256$ . This final, finest refinement case creates many levels in-between  $p = 1$  and  $p = 256$ , making it an ideal test case for the efficiency of the implementation.

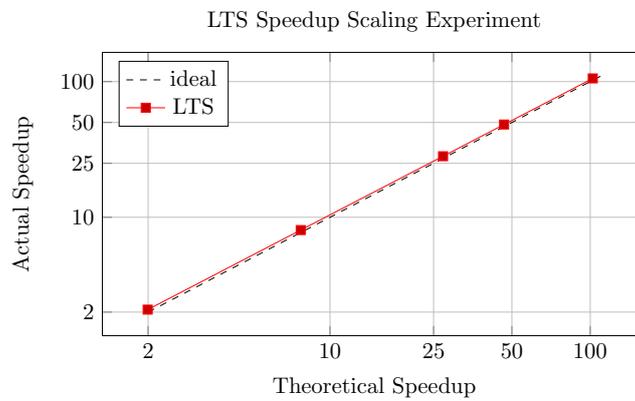


Figure 3.8. LTS scaling test on a 300,000 element (19M DOFs) mesh with a center refinement that can be scaled from 2x–100x speedup over the non-LTS version.

Performance experiments from these meshes that compare the LTS and non-LTS versions of the code are depicted in Fig. 3.8, where we see that the actual application speedup matches the ideal speedup almost perfectly. By ensuring that the LTS operations scale with the number of elements in each LTS level, we have achieved excellent efficiency. For example on the 100x speedup mesh, the finest level has  $p = 256$  and only 1,136 elements (compared to 298,000 on the coarsest level with  $p = 1$ ). Without the analysis in Sec. 3.5, the time-stepping operations for the lowest level will be performed 256 times per global step on all 19M DOFs. The near perfect efficiency shown in the figure demonstrates that, for a single-threaded application, the overhead introduced by an efficient LTS implementation is minimal. However, real-world applications also demand excellent *parallel* scaling performance as well.

## 3.8 Conclusions

We have presented the LTS-Newmark scheme and its high performance implementation for the large-scale simulation of wave propagation. Expanding on previous LTS work, we provide the ability to utilize multiple refinement levels, yielding more performance than a simpler 2-level scheme. We additionally provide the algorithmic details necessary to efficiently implement the scheme in a continuous finite-element spatial discretization. By using the halo selections  $\mathbf{R}$ , we can extract the minimal set of operations necessary for LTS. Without this, the LTS scaling up to 100x from the previous Sec. 3.7.4 would not be possible.

These efficiency considerations also extend from the CPU version to the GPU version (Ch. 2). This allows us to compound the speedup over the reference CPU version by combining LTS and GPU speedup, yielding one or two orders of magnitude of performance over a standard explicit CPU version without LTS. Of course this speedup depends on the mesh in question, which is generally constructed for a particular application or experiment.

However, the excellent performance shown thus far has only used a single CPU or GPU, without consideration for running the simulations across hundreds to thousands of CPUs or GPUs, allowing for larger domains from both a memory and simulation-time perspective. However LTS-Newmark, particularly the multi-level scheme, significantly changes the standard approach to parallelism taken by most finite-element wave-propagation packages. Solving this multi-CPU/GPU problem is the focus of the next chapter and will allow the code to scale to thousands of CPUs for extremely large problem sizes with millions of elements and billions of degrees of freedom.

## **Chapter 4**

# **Load-Balanced Local Time Stepping at Scale**

## 4.1 Introduction

In Ch. 3, we introduced the explicit LTS-Newmark scheme. Critical to the usefulness in seismology applications, this analysis included details on the efficient implementation of LTS-Newmark in SPEC3D [Peter et al., 2011] for hexahedral meshes with embedded refinements that create a theoretical LTS speedup of up to 100x (3.46), which the code can indeed achieve. However, these speedup experiments were only done using a single CPU or GPU. Practical seismological modeling and simulation require domain sizes and resolutions such that the resulting finite-element meshes are simply too large for a single CPU (or GPU) in both time-to-solution and memory requirements.

Recall that LTS methods have been introduced to localize the time-step size to the element size, and can minimize the effect of these small elements on the overall performance. These LTS methods, however, create a load-balancing problem when the simulation is run in parallel across many processors. For example, Minisini et al. [2013] combined a two-level leapfrog scheme with a DG finite-element discretization to model salt interfaces, a common imaging application used in exploration geophysics. These interfaces, in a two-level only element distribution, were good for a speedup of 4x. A multi-level LTS scheme, such as our LTS-Newmark scheme, would have likely supported another 1.5–2x performance increase for this mesh. However, besides shared memory parallelism, they did not implement a multinode parallel solver due to the load-balancing challenges.

This chapter is particularly focused on solving this load-balancing problem and presents several successful solutions using multiconstraint graph and hypergraph partitioning algorithms. We compare these algorithms for a variety of examples on both CPU and GPU clusters using our newly developed high-performance implementation from Ch. 3.

## 4.2 The Partitioning Problem

Real-world seismology problems are too big for a single machine in both memory and compute cost and thus require parallelization. Packages such as SPEC3D follow the traditional parallelization approach, where a finite-element mesh is partitioned using a tool such as SCOTCH [Chevalier and Pellegrini, 2008] or MeTiS [Karypis and Kumar, 1998]. In the standard approach, the stiffness matrix contribution is computed for partition boundaries first, which are then exchanged using asynchronous MPI communications. We follow this approach in

the GPU implementation: the asynchronous overlapping is repeated for the required GPU-CPU memory copies as well (see Sec. 2.4).

Due to LTS, however, some elements take more steps and thus require more work than other elements. An unmodified partitioner will lead to a parallelization that is unbalanced, reducing overall performance. Standard partitioning packages such as SCOTCH allow weighted elements and produce partitions that are weight balanced according to these inputs. Unfortunately, this weighting technique also has a balancing problem, due to the way an LTS scheme steps through the elements in time.

Indeed LTS introduces an additional balancing constraint to the partitioning of the mesh. As seen in the multilevel algorithm, the steps of the LTS algorithm proceed recursively through the lower levels, until finally taking a step on the coarsest global level. We define an *LTS cycle* as the work needed to take all steps at every level until the coarsest level takes a step of size  $\Delta t$ .

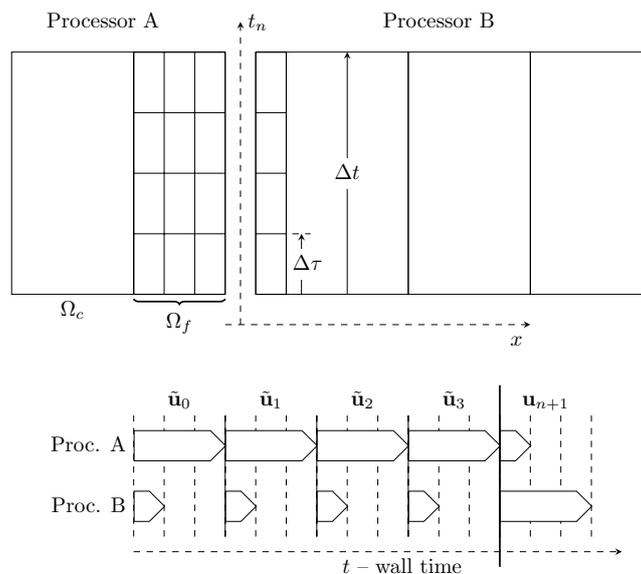


Figure 4.1. A timeline of a 1D mesh with two partitions that are balanced without consideration for LTS. The partition for processor A has three fine elements, and a single coarse element, whereas partition B has only a single fine element and three coarse elements. The top graphic shows how many steps per  $\Delta t$  each element must take, and the bottom graphic depicts a runtime profile showing the fine-level time steps  $\tilde{u}_m$  and the required synchronization between partitions at every step. We note that processor B stalls waiting for processor A (and vice versa) due to the coarse-fine imbalance.

For instance, Fig. 4.1 shows a 1D time-stepping diagram with two partitions from a standard partitioner, and the corresponding unbalanced timeline. Each of the four fine-level steps requires synchronization between the partitions. Furthermore, since the partition A has three times more fine elements than partition B, processor A will take three times longer than processor B to complete a single fine-level step  $\Delta\tau$ . Once the fine level ( $\Omega_f$ ) completes, processor A will stall waiting for processor B due to the imbalance of the coarse elements. This imbalance on each refinement level across only two processors will drastically reduce the hard-won efficiency seen in the sequential version. Furthermore, the parallelization should additionally consider the increased communication requirements between elements with  $p > 1$ .

A scalable parallelization solution is thus critical to the success of LTS in real-world applications. For the scope of this thesis, we present a solution relying on partitioning tools that provides good scalability and can be easily adopted into existing code bases that use traditional partitioning techniques. We call this solution *p-level balanced partitioning*, as it attempts to balance the load across each level of refinement (p-level) using existing partitioning tools by partitioning each p-level equally across processors.

Previously, for the two-level scheme proposed by Godel et al. [2010], in order to allow for multiple GPUs, the partitioning is restricted to only cut across coarse ( $p = 1$ ) elements, ensuring that MPI synchronization is only required every  $\Delta t$  and not for any substeps. By merging fine-level elements in the mesh's graph representation and correspondingly weighting them by cost, a partitioner like SCOTCH or MeTiS is able to ensure good load balancing while guaranteeing that cuts occur only between coarse elements. We also considered this approach, but rejected it because it inherently limits the scalability with an artificially high lower limit on the number of elements per partition. At some point the partition cannot be split further without cutting across fine-level ( $p > 1$ ) elements. Despite this limitation, we do briefly explore this partitioning strategy in Sec. 4.3.7.

In the multilevel ADER-DG scheme [Dumbser et al., 2007], elements of a similar time-step size are grouped (analogous to our  $p$ -levels) and partitioned individually and remerged with the other groups to provide a single partition per processor. This is very similar to our SCOTCH-P approach to be introduced in Sec. 4.2.2 and motivated the use of a multi-constraint partitioner to do this in a single step without the need to remerge partitions.

With our initial partitioning goals set, we can now discuss how LTS further changes the partitioning cost, and compare graph and hypergraph partitioning approaches, including the communication and multiconstraint modeling for each type. Then, we introduce several implementations using a variety of parti-

tioning libraries (compared later in Section 4.3).

### 4.2.1 LTS-Partition models: graphs and hypergraphs

To design an LTS-aware partitioning algorithm, we need to model the LTS requirements in terms of load balance and communication costs. Ensuring that each processor is equally loaded for each  $\Delta t/p$  substep is vital for the parallel efficiency of the implementation. Existing graph partitioning tools can balance work between partitions by weighting the graph vertices, which can be used to balance cheaper acoustic domains with more expensive elastic ones, for example. However, as noted, the recursive nature of LTS requires additional balancing inputs, one for each level.

The standard graph and hypergraph partitioning problems (which are NP-complete) ask for a partition of the vertices of the given (hyper)graph in a given number  $K$  of nonempty, disjoint sets. The partitioning constraint of both problems is to achieve a balance on the part weights, usually defined as the total weight of the constituting vertices. The partitioning objective, called cut size, is to reduce the number or weight of the edges having vertices in different parts in the graph partitioning problem. In the hypergraph partitioning problem a function of the hyperedges straddling the partition boundaries is the objective function.

A recent variant of the standard graph and hypergraph partitioning problem blends multiple balance constraints, which is therefore called the *multi-constraint graph and hypergraph partitioning problem* [Aykanat et al., 2008]. In this problem, each vertex has a vector of weights. We use  $w[v, i]$  to denote the  $P$  weights associated with the vertex  $v$ , for  $i = 1, \dots, P$ . For a vertex set  $U$ , we use  $W[U, i]$  to denote the sum of the  $i$ th weights of vertices in  $U$ , i.e.,  $W[U, i] = \sum_{u \in U} w[u, i]$ . In this setting, the partitioning constraint is to satisfy a balance criterion for each  $i = 1, \dots, P$ :

$$W[V_k, i] \leq (1 + \varepsilon) \frac{W[V, i]}{K} \text{ for } k = 1, \dots, K, \quad (4.1)$$

for an allowed imbalance  $\varepsilon$  (the partitioning objective remains the same for both of the two partitioning problems).

Besides requiring more computational work, elements in a finer level (e.g.,  $\Delta t/2$ ) also create  $p$ -times more communication volume when split between processors. This higher cost is visualized in Fig. 4.2, an example (higher-order) 2D finite-element mesh with 9 nodes per element. The black nodes are in a higher  $p$ -level ( $p = 2$ ) and thus require more communication when cut. The bordering

gray nodes are a type of “halo” due to the LTS algorithm for continuous elements, and also require updates on each  $\Delta t/2$  step. Visualized in each of the lower meshes, the communication cost associated with the three possible partition cuts is shown to highlight how LTS changes the cost associated with the cuts along the different levels of the mesh. In order to partition our mesh across processors, we turn to graph and hypergraph partitioning tools.

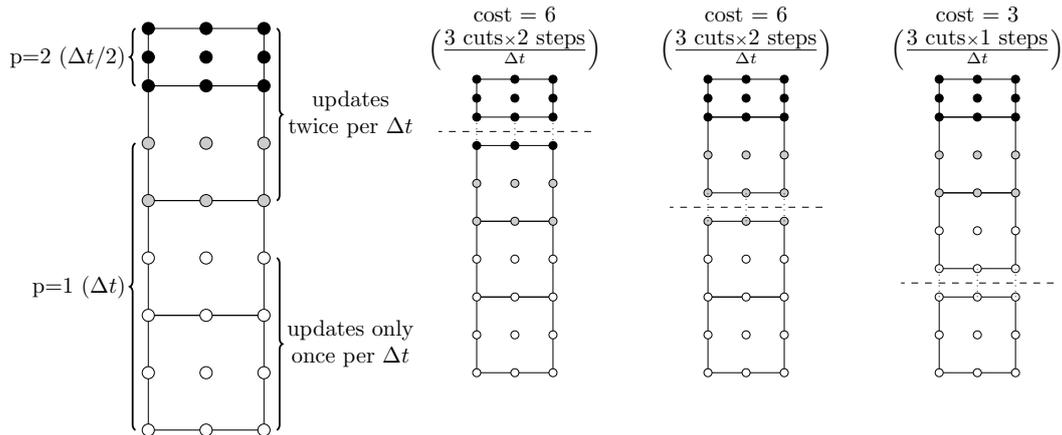


Figure 4.2. A 2D finite-element mesh with four elements depicting the partitioning cost for each of three possible cuts. The nodes in black and the nodes in gray are updated at every  $\Delta t/2$  step. A cut across black or gray nodes will require 2 synchronizations for every LTS cycle ( $\Delta t$  step).

## Graphs

A mesh itself is a bipartite graph [Chevalier and Pellegrini, 2008], where elements are defined by their corner vertices (nodes), and vertices are connected to multiple elements. For a standard graph partitioning tool such as SCOTCH or MeTiS, we first create the mesh’s dual graph, which represents the connection of mesh elements across faces as seen in Fig. 4.3. The dual graph only accounts for the communication requirements of nodes on an element’s face. Corner nodes, on the other hand, are connected to multiple elements and are not modeled by this simply connected graph.

In the absence of LTS, vertices and edges are generally unweighted. For tools that support this, correct load balancing for LTS is only achievable through the multiconstraint approach previously mentioned. Each vertex is assigned the weight vector  $w[\gamma, i]$  corresponding to the load of the associated element at level  $i$ , which should be balanced by the partitioning library appropriately.

The edges of the graph are also weighted according to the  $p$ -level, with the weight of the edge set to the maximum value of  $p$  for the two connected vertices. This edge weighting can only approximate the cost detailed in Fig. 4.2. To build a fully accurate cost model, we require a hypergraph, which allows the edges to connect all relevant vertices.

## Hypergraphs

Here we first formally describe the standard hypergraph model for the (typical) finite-element mesh and the associated partitioning problem, to set the scene for an accurate hypergraph model for LTS. Recall that a hypergraph is an ordered pair  $H = (V, N)$  consisting of a set  $V$  of vertices and a set  $N$  of hyperedges (or nets), where each hyperedge is a subset of the vertex set  $V$ , and that the vertices can have weights and the hyperedges can have costs. We again use  $w[v]$  and  $w[U] = \sum_{u \in U} w[u]$  to denote the weight of a vertex  $v$  and the sum of the weights of vertices in the vertex set  $U$ , respectively. We use  $c[h]$  to denote the cost of a hyperedge  $h$ .

In the hypergraph model of a mesh, the vertices correspond to elements, and the hyperedges correspond to the corner nodes. Each corner node defines a hyperedge and connects all elements (vertices) touching it. Thus a corner node might connect 4 or more elements in two dimensions, and 8 or more elements in three dimensions. A simple 2D example is shown in Fig. 4.3. The diagram shows a 4-element rectangular mesh and its corresponding dual graph and hypergraph. In the case where all 4 elements are in a separate partition, the dual graph will only count 4 edges in each cut, where the hypergraph will add the additional cuts between all 4 elements due to the central node, modeling the additional required MPI communication accurately.

$\Pi = \{V_1, \dots, V_K\}$  is a  $K$ -way vertex partition of a given hypergraph  $H = (V, N)$ , if each part is a nonempty subset of  $V$ , parts are pairwise disjoint (that is,  $V_i \cap V_j = \emptyset$  for  $i \neq j$ ), and collectively exhaustive (that is,  $V = \bigcup V_i$ ). A  $K$ -way vertex partition  $\Pi$  is *balanced* if (4.1) holds for  $V_1, \dots, V_K$  with  $P = 1$ . The cost of a vertex partition  $\Pi$  of a given hypergraph  $H = (V, N)$  is called the *cut size*. It measures the degree of the spread of the hyperedges that have vertices in different parts, weighted according to the cost of the hyperedges. We formally define the weighted cut size as the following,

$$\text{cutsize}(\Pi) = \sum_{h \in N} c[h](\lambda_h - 1), \quad (4.2)$$

where  $\lambda_h$  is the number of parts in which the hyperedge  $h$  has vertices, recalling

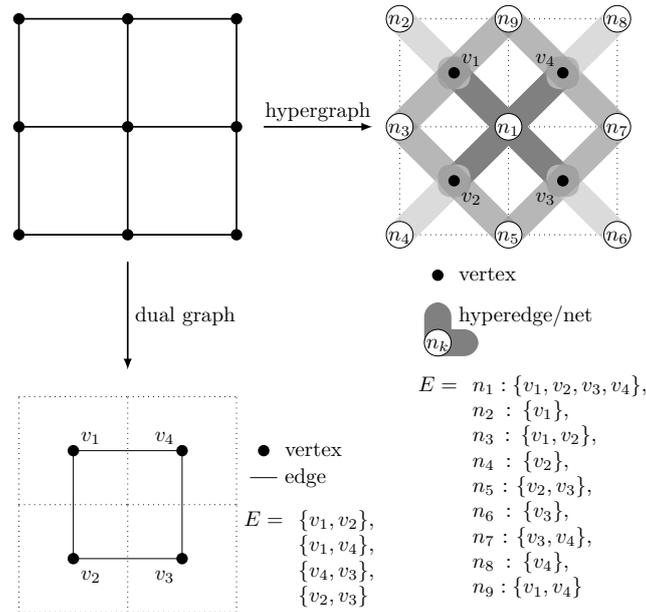


Figure 4.3. Graph vs. hypergraph representations of a 2D finite-element mesh. The traditional dual graph can only model the connection between elements sharing a face, where the hypergraph models the connection between all elements that share a node (e.g., when four elements share a corner).

$c[h]$  as the hyperedge weight. This definition simply counts weighted (hyperedge) cuts across partitions.

Given a hypergraph  $H = (V, N)$ , an integer  $K$ , and an imbalance parameter  $\varepsilon$ , the hypergraph partitioning (HP) problem asks for a balanced (4.1),  $K$ -way vertex partition  $\Pi$ , with the minimum cut size (4.2).

We now propose a hypergraph model for partitioning meshes for load-balanced LTS computations with reduced communication cost. For a given finite-element mesh, we create a hypergraph  $H = (V, N)$  in two steps. In the first step, we define vertices and their weights so that a balanced partitioning of the vertices will correspond to a balanced computational load distribution among processors at all LTS levels. In the second step, we define hyperedges and their costs such that the total volume of communication in an LTS cycle will exactly match the cut size (4.2), when the elements are partitioned according to the vertex partitions.

The vertices and their weights in  $H$  are defined as follows. Each element of the mesh is uniquely represented by a vertex in  $V$ . As an element belongs to a level, the corresponding vertex in  $H$  can be said to belong to a level. We associate a weight vector of size  $P$ , where  $P$  is the number of levels, with each vertex. In

this setting, the vertex weight vector  $w[v, i]$  is set to one for  $i$  corresponding to the level of the associated element, and all other weight coordinates are set to zero. Once these are set, we partition the vertices of  $H$  into  $K$  parts, where the balance criteria (4.1) are satisfied for all weight coordinates. After assigning each part to a processor, we obtain a (hopefully) load-balanced partitioning such that the computational work is evenly distributed among processors across all LTS levels.

The hyperedges and their costs in  $H$  are defined as follows. For each node  $n$  of the mesh, we create a hyperedge  $h_n$ , where  $h_n$  contains vertices corresponding to the elements containing the node  $n$ . We put a copy of this hyperedge to the hypergraph for each element containing the node  $n$ . We use  $h_n^{(1)}, \dots, h_n^{(e)}$  to refer to the  $e$  copies of the hyperedge, where  $e = |\text{elmnts}(n)|$  is the number of elements containing  $n$ . Thus, when the set of elements  $\text{elmnts}(n)$  is assigned to  $\lambda_n$  different processors, for each element in  $\text{elmnts}(n)$ ,  $\lambda_n - 1$  messages need to be delivered from the owner of the element to other processors. Now, as each element belongs to a particular LTS level, the messages should be delivered according to the step size of the level. Therefore, for each hyperedge,  $c[h_n^{(i)}]$  is set to  $\chi$ , where  $\chi$  is the level of the  $i$ th element in the set  $\text{elmnts}(n)$ . With this hyperedge and cost definitions,  $\sum_{i=1}^e c[h_n^{(i)}](\lambda_n - 1)$  cost is added to the cut size, when the  $e$  elements containing the node  $n$  are partitioned among  $\lambda_n - 1$  processors. Since the total volume of communication can be computed by the sum of the volume of communication per element, the cut size (4.2) represents the total volume of communication in an LTS cycle accurately. A simplification is possible here. Since all hyperedges  $h_n^{(1)}, \dots, h_n^{(e)}$  associated with the node  $n$  have the same set of vertices, we can represent them with a single hyperedge  $h'_n$  with  $c[h'_n] = \sum_{i=1}^e c[h_n^{(i)}]$ . Then, the number of hyperedges is reduced to the number of nodes in the mesh without any loss in the correspondence between the cut size and the total volume of communication.

### 4.2.2 Partitioning algorithms for LTS

Once the graph and hypergraph models are defined, we can develop methods to partition the mesh based on those models. We examined the following four techniques:

**SCOTCH:** This is the standard graph partitioner which is used in SPEC-FEM3D. It performs a standard partitioning, but assigns a single weight to each element according to the p-level, such that each partition will have equal work (measured over a global  $\Delta t$  step), but will be unbalanced for substeps taken at different

LTS levels. This provides a baseline to measure the relative success of a multi-constraint setup.

**SCOTCH-P:** SCOTCH itself provides only single-constraint partitioning. We propose the following approach to use SCOTCH beyond the baseline. Each  $p$ -level is partitioned separately among all processors using the standard SCOTCH routines, so that the partitions at all levels have a balanced load. We then map exactly one partition from each level to a single processor so that the processors have balanced load across all levels. While mapping partitions from each level, we greedily couple each partition from level 1 to the best available partition from level 2, and so on. One could experiment with more efficient mapping methods (based on weighted graph matchings), but we reserve this for future work.

**MeTiS:** As of version 5.0, MeTiS can perform a multiconstraint graph partition with weighted edges, attempting to balance  $p$ -levels and reduce the edge cut as an upper bound to the total communication volume simultaneously.

**PaToH:** A hypergraph partitioner [Çatalyürek and Aykanat, 1999] which performs a multiconstraint partitioning with weighted hyperedges to accurately minimize the total communication volume.

## 4.3 Performance Experiments

As noted, we integrated the LTS-Newmark algorithm into the seismology software package SPECFEM3D, which was explored in detail in Ch. 2. Primarily written in Fortran95, the CPU version remains a purely MPI-based code, such that a typical simulation is run using a single MPI rank per processor. In contrast, the GPU version typically runs a single MPI rank per GPU, which is commonly 1 or 2 GPUs per supercomputing node.

### 4.3.1 Application mesh benchmarks

In order to test our LTS implementation at large scale, we assembled three test benchmark hexahedral meshes that replicate refinement seen in real-world applications. In Fig. 4.4 we show smaller examples of these benchmarks, with colored  $p$ -levels. In Table 4.1 we see the size and theoretical speedup (3.25) (from Ch. 3) of each mesh used for testing. The *trench* mesh is designed to model a long strip of refinement, a common problem seen in several application

meshes, especially where two internal topographies meet and produce a long row of pinched elements. The *embedding* mesh is the simplest possible example of refinement and models any localized small-scale feature. The *crust* example models topography and large-scale surface features. Each of these benchmarks

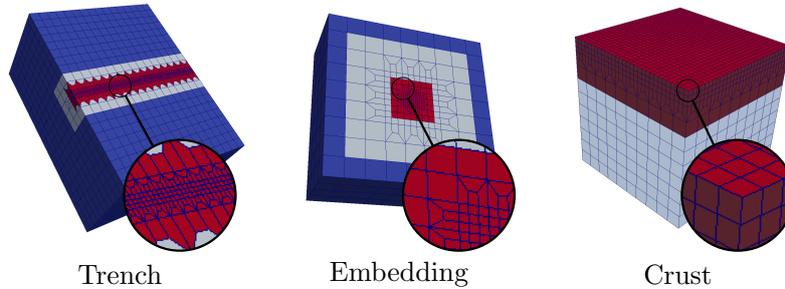


Figure 4.4. Small examples of the four benchmark meshes used to compare and test the performance of each partitioning scheme along with LTS implementation performance. Actual performance benchmarks were conducted on larger examples of these meshes. Smallest p-level elements colored in red, mid-sized in gray, and largest in blue.

Mesh	# elements	# DOFs	Theor. LTS speedup	# of levels
Trench	2.5M	170M	6.7	4
Trench Big	26M	1.7B	21.7	6
Embedding	1.2M	78M	7.9	4
Crust	2.9M	190M	1.9	2

Table 4.1. Benchmark meshes in detail. The fourth-order elements have 125 nodes per element, increasing the total DOFs by almost two orders of magnitude relative to the number of elements.

can, in principle, be scaled to any size and any level of LTS speedup. However, the crust mesh or other examples with topography, for instance, are limited in LTS speedup because of the large number of small elements on the surface, making it impossible to increase the ratio of large to small elements without making an unrealistically tall and skinny mesh. In this case, tetrahedral elements are better able to conform to a desired topography and body element size, and thus can yield a higher expected LTS speedup. However, as previously noted, we are currently limited to hexahedral elements in the absence of a stable and efficient quadrature rule for tetrahedra to allow for a diagonal mass matrix.

### 4.3.2 Partitioning results

We visualize each partitioning type in Fig. 4.5, where we see the balancing across levels for the PaToH, MeTiS, and SCOTCH-P schemes, whereas the original, single-weighted SCOTCH partitioning is only balanced across an entire LTS cycle, creating a load imbalance at each substep. In order to compare and quantify each partitioning implementation, we conducted load balance, communication cost, and application performance experiments using large versions of each type of mesh depicted in Fig. 4.4.

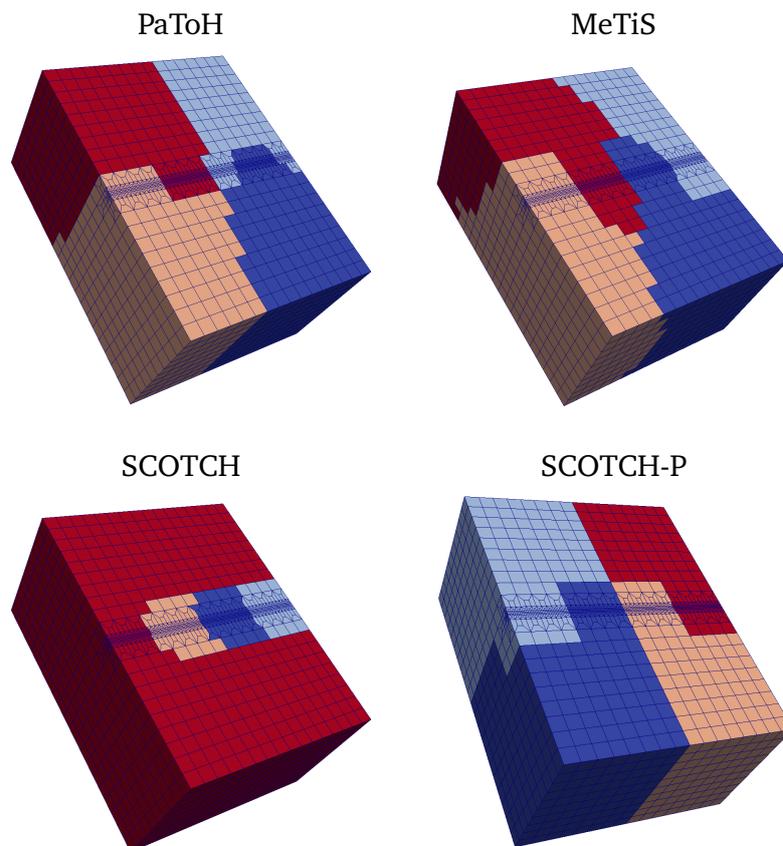


Figure 4.5. All partitioning tools on example trench mesh with 4 partitions (partition seen by color). Note that SCOTCH (incorrectly) only balances work per LTS cycle, whereas the others balance each level correctly.

Obviously, the most important goal of each partitioner is to produce the highest possible application performance. However, it is useful to compare partitioning performance in terms of graph cut, total communication volume, and load balance to help explain and understand application performance differences.

For example, the PaToH partitioning library exposes many parameters, including the desired final imbalance (`final_imbal`), which affects the trade-off between communication cost (cuts along hyperedges) and the balance between p-levels across partitions.

We are interested in two metrics, starting with load imbalance defined as

$$\text{load imbalance \%} = \frac{(\text{max load}) - (\text{min load})}{(\text{max load})} \times 100, \quad (4.3)$$

where load is the estimated computational load per partition. We define load as the sum of graph vertices each weighted by their respective p-level refinement  $p$  (each element has approximate computational cost  $p$ ). This is measured for both the entire mesh, and across p-levels.

Using the 2.5M-element trench mesh, we compare the load imbalance across the MeTiS, PaToH, and SCOTCH-P partitioners in Table 4.2. The imbalance

# of parts	Load imbalance			
	MeTiS	PaToH 0.05	PaToH 0.01	SCOTCH-P
16	34%	11%	2%	6%
32	88%	17%	5%	6%
64	89%	19%	7%	7%

Table 4.2. Total work load imbalance (4.3) for MeTiS, PaToH, and SCOTCH-P partitioners on 2.5M mesh. PaToH is additionally compared with two values of parameter `final_imbal=0.05, 0.01`.

table highlights several important points. The MeTiS multiconstraint partitioner is currently not able to maintain an optimal balance across levels, where the PaToH partitioner does manage this balance, where the `final_imbal` parameter can be used to improve the balance at the cost of additional communications.

The second critical metric is the weighted graph cut and the total MPI-communications volume. The traditional partitioners MeTiS and SCOTCH-P both utilize weighted graph cut metrics, as opposed to PaToH, which optimizes the weighted hypergraph cut, which accurately models total communications volume as noted in Fig. 4.3. Again using the 2.5M-element trench mesh, we compare graph cut and total communication volume metrics across each partitioner in Table 4.3.

Although MeTiS is able to produce a better *graph cut*, the *MPI volume* is better optimized by PaToH and its more accurate hypergraph representation. However, as noted, when we examine load imbalance, MeTiS does not compare favorably.

# of parts	MeTiS		PaToH 0.05	
	Graph cut	MPI volume	Graph cut	MPI volume
16	$1.4 \times 10^6$	$1.0 \times 10^7$	$1.8 \times 10^6$	$1.1 \times 10^7$
32	$2.4 \times 10^6$	$2.0 \times 10^7$	$2.9 \times 10^6$	$1.8 \times 10^7$
64	$3.5 \times 10^6$	$3.0 \times 10^7$	$4.2 \times 10^6$	$2.6 \times 10^7$
# of parts	SCOTCH-P		PaToH 0.01	
	Graph cut	MPI volume	Graph cut	MPI volume
16	$1.9 \times 10^6$	$1.3 \times 10^7$	$1.0 \times 10^6$	$1.0 \times 10^7$
32	$3.1 \times 10^6$	$2.1 \times 10^7$	$2.3 \times 10^6$	$1.6 \times 10^7$
64	$4.7 \times 10^6$	$3.3 \times 10^7$	$3.4 \times 10^6$	$2.3 \times 10^7$

Table 4.3. Table comparing communication cost metrics between MeTiS, PaToH (with `final_imbal=0.05,0.01`), and SCOTCH-P for refinement trench mesh with 2.5M elements. *Graph cut* is the weighted cost of cut edges for the simple graph, and *MPI volume* is the total MPI-communications volume per LTS cycle.

We note that SCOTCH-P is able to beat both MeTiS and PaToH in terms of communication costs while maintaining a better load balance. The simple greedy reorganization used by SCOTCH-P after the p-level partitioning seems to work extremely well for the mesh examples we tested. From these partitioning experiments, we expect SCOTCH-P and PaToH to perform well in the application performance experiments in the next section, where we also can evaluate the effect of the load imbalance and communication cost trade-off for PaToH.

### 4.3.3 CPU and GPU performance results

We ran benchmarks on the large CPU and GPU cluster *Piz Daint*. Each compute node is powered by a single 8-core Intel E5-2670, and a single NVIDIA Tesla K20X, where the CPU version runs 1 process per core (8 per node) and the GPU version runs 1 process per GPU (1 per node), and we compare performance on a node-to-node basis. Because LTS improves the efficiency of the time-stepping method, we must evaluate performance in terms of the wall-clock time to compute a fixed amount of simulation time. More simply, we are interested in the wall-clock time (in seconds) it takes to simulate, e.g.,  $T = 100$  s of wave propagation. A non-LTS scheme is forced to take the globally smallest time step ( $\Delta t_{\min} = \Delta t / p_{\max}$ ), and we measure the time it takes to simulate  $T / (\Delta t_{\min})$  steps. LTS, on the other hand, takes steps of different sizes, and globally synchronizes every  $\Delta t$ , such that every  $\Delta t$  of simulated time is less expensive com-

pared to the non-LTS scheme. Thus performance is measured as [simulated time]/[wall-clock time] (s/s), however, we opt instead to present our results normalized (relative) to the non-LTS (reference) CPU version at, e.g., 16 nodes (128 cores). This presents the total speedup achieved by LTS, the non-LTS GPU version, and the LTS GPU version.

We also differentiate between simple scaling efficiency, LTS efficiency, and the LTS-scaling efficiency. Listed in each of the performance scaling figures, we list the scaling efficiency of the non-LTS CPU and GPU versions, which simply compares against an ideally scaling code starting at 16 nodes. For the LTS case, LTS scaling efficiency is compared against an ideal LTS code that starts at the speedup predicted by the speedup model (Ch. 3) (3.25), and achieves perfect scaling. The CPU version (at, e.g., 16 nodes) typically achieves 100% LTS efficiency, where the GPU version at 16 nodes might only achieve 86% LTS efficiency relative to the predicted speedup using the non-LTS GPU version.

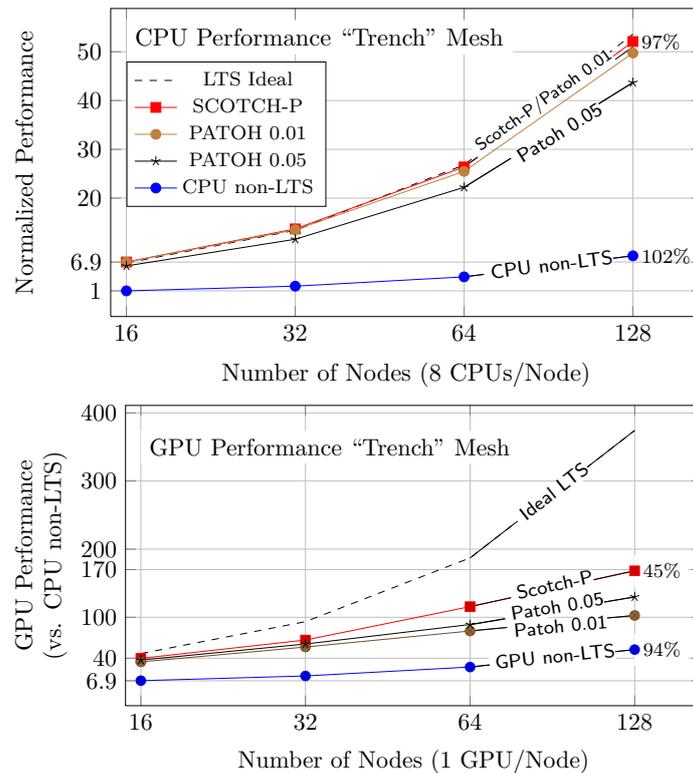


Figure 4.6. Performance results of the 2.5M-element trench mesh comparing the different LTS partitioning strategies (predicted speedup = 6.7x) using CPUs (top) and GPUs (bottom), relative to the reference (non-LTS) CPU code on 16 nodes.

Using the 2.5M-element trench model, we evaluated performance from 16 to 128 nodes, with either CPUs or GPUs. The speedup of the LTS version, relative to the original (non-LTS) SPEC-FEM3D CPU version is highlighted in Fig. 4.6. Also shown is the ideal LTS scaling curve, which assumes perfect LTS efficiency and scalability. The percentages listed next to LTS-CPU (97%), LTS-GPU (45%), non-LTS CPU (102%), and non-LTS GPU (94%) are the LTS and non-LTS scaling efficiencies relative to their respective ideal scaling curves.

Both PaToH and SCOTCH-P partitioning methods perform very well up to 1,024 cores. Both the non-LTS and LTS CPU versions achieve very high scaling efficiency up to at least 128 nodes, which profiling indicates is partially a result of cache performance improving as the partitions grow smaller, which we will analyze in more detail in Sec 4.3.4. We also compared PaToH performance between the `final_imbal` runtime parameter, where we note that load balance is the correct trade-off for CPU performance.

For the trench example, we also consider GPU performance, as compared to the reference CPU version at 16 nodes. The non-LTS GPU version achieves a speedup of 6.9x over the non-LTS CPU version, and the LTS-GPU with SCOTCH-P starts at 84% LTS efficiency, but is not able to maintain more than 80% efficiency past 32 nodes. Profiling indicates that this scaling inefficiency is mostly due to kernel setup and launch overhead for the very small number of elements in the finer p-levels in each partition. This strong-scaling inefficiency is an expected limitation of this parallelization method, and is most obvious with the GPU version, where the kernel setup and launch overhead dominates the runtime as the number of elements in the smallest levels shrinks.

We further investigate the performance using the “embedding” mesh in Fig. 4.7. We see similar results to the trench mesh, where SCOTCH-P performs best, followed by the better balanced PaToH. At 16 nodes, SCOTCH-P is able to achieve 95% of the theoretical LTS speedup of 7.9x over the non-LTS CPU version, also at 16 nodes. We again see the superlinear scaling attributed to improved cache performance.

We now turn to the “crust” mesh, which is limited to a 1.9x theoretical LTS speedup, due to the large number of small elements along the surface. Seen in Fig. 4.8, the PaToH and SCOTCH-P partitioning options perform comparably and achieve 96% scaling efficiency at 128 nodes (1024 processors), which is especially important given the limited speedup available for a mesh of this type. Again we see the importance of the stricter load-balancing constraint for PaToH.

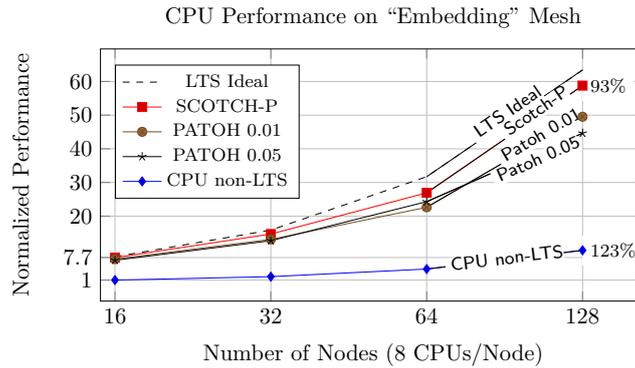


Figure 4.7. Performance results on 1.2M-element embedding mesh with 7.9x theoretical speedup. We compare SCOTCH-P and PaToH partitioners, including the `final_imbal` PaToH configuration parameter.

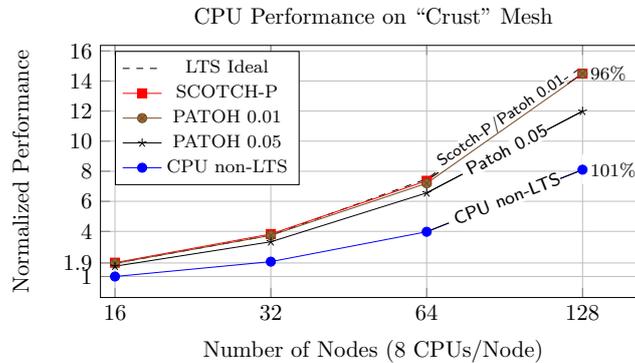


Figure 4.8. Performance results on 2.9M-element crustal mesh with 1.9x theoretical speedup. We note that the PaToH 0.01 and SCOTCH-P scaling curves are nearly identical.

#### 4.3.4 Cache performance

As noted, the scaling performance of the reference version for the trench mesh exhibits superlinear speedup. Although the finite-element mesh is quite regular, the layout of the elements and nodal degrees of freedom in memory has never been optimized. Using the Cray performance tool `craypat` to gather a D1+D2 cache utilization metric (hits of L1+L2 data cache), we conducted an experiment using both the reference and LTS versions from 16 to 128 nodes as seen in Fig. 4.9.

From Fig. 4.9, we see the expected higher cache use for the reference version coinciding with the superlinear scaling performance. We also note that the LTS

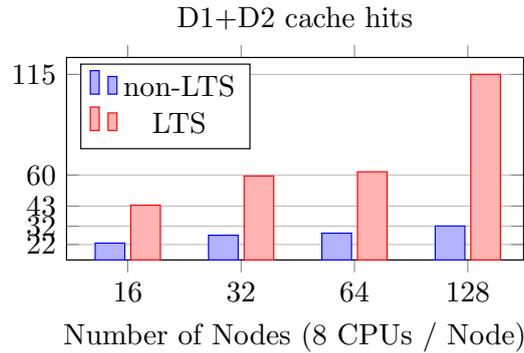


Figure 4.9. CPU D1 + D2 (L1 + L2) level cache hit metric for non-LTS and LTS versions on trench mesh. More hits means greater cache utilization.

version of the code achieves an even greater utilization of cache; the nodal DOFs are grouped by  $p$ -level in order to utilize vector operations, which additionally improves cache performance. The LTS algorithm also naturally improves locality because the lowest  $p$ -levels contain a small amount of elements (in the ideal case) and require  $p$  computations per global  $\Delta t$ , such that many of the memory locations will remain in cache for each  $\Delta t/p$  step. We believe that this excellent cache utilization allows the CPU LTS code to overcome the LTS overhead resulting in an efficient scaling. Unfortunately, the GPU version is unable to benefit from these cache advantages, as shown by its lower scaling efficiency.

### 4.3.5 Large example

The benchmark meshes seen thus far were designed to mirror modestly sized wave-propagation problems across several applications of interest. In the field of computational seismology, there are often several levels of parallelism to exploit, as many real-world applications require many independent source simulations. However, as the problems and computers get bigger, it is important to test large meshes on a large number of processors in order to find bottlenecks in the code that may not have been visible before.

We extended the “trench” example by an order of magnitude to include 26M elements (vs. 2.4M), with two additional refinement layers to increase the theoretical speedup to 21.3x. Figure 4.10 shows scaling experiments from 128 to 1024 nodes (1024 to 8192 processors, resp.) using the SCOTCH-P partitioner. We see that this large example scales very well, despite the high large number of processors and higher LTS speedup as compared to the other examples.

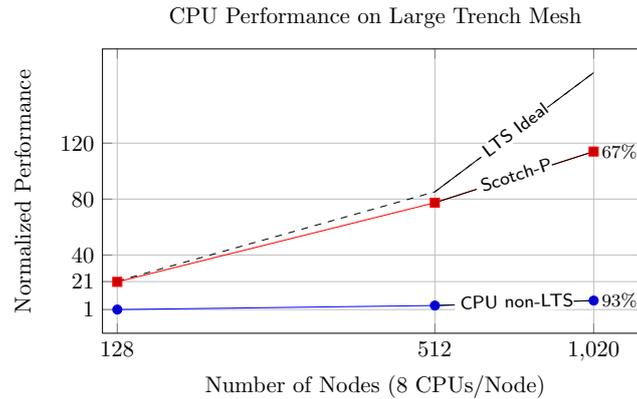


Figure 4.10. Performance results on largest 26M mesh for up to 8192 processors across 1,024 nodes. The LTS scaling efficiency starts at nearly 100% and remains excellent until 512 nodes (4,192 processors), but drops off to 67% at 1,024 nodes (8,192 processors).

### 4.3.6 Application example: Tohoku

With the excellent performance on large synthetic benchmarks, we turn to a real-world example for further performance validation. For many seismic applications, a mesh is designed to support a minimum wavelength, which is usually dependent on the resolution of the velocity structure of the medium. However, in order to represent internal or external structures, it is common that lines or regions of small elements occur due to meshing difficulties or the dimensions of the structure. Seen in Fig. 4.11, our application mesh is designed to model the Tohoku fault in Japan, the source of the magnitude 9.0 earthquake in 2011. With 7.5M elements, it is relatively large for a SPEC3D regional simulation. As the images show, the mesh as well as the element distribution show how the fault creates an ideal situation for LTS — the small elements along the fault strongly impact the efficiency of a standard explicit time-stepping scheme. The predicted speedup of this mesh is 4.1x, which is less than most of our synthetic benchmarks, but represents a nontrivial performance loss nonetheless. Additionally, this example can be run on GPUs as well with potential for almost 30x overall performance increase.

Traditionally in SPEC3D and other seismic simulation codes, earthquakes are modeled as point sources, or possibly a collection of point sources. These sources simply prescribe the slip at static mesh points, and do not represent the dynamic triggering of a true fault. Seismologists are now trying to model the earthquake as a dynamic fault rupture, which requires solving dynamic rupture

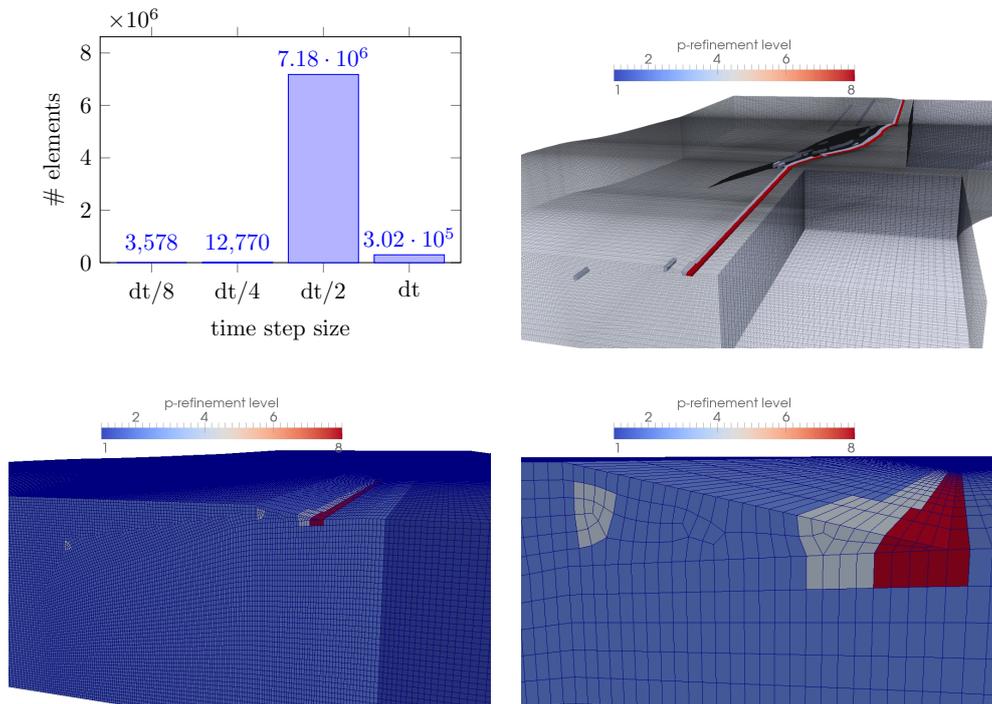


Figure 4.11. Tohoku mesh with 7.5M elements and a predicted speedup of 4.1 with the given element distribution.

physics on traditionally static mesh. In order to implement this in SPECFEM3D, Galvez et al. [2010] require that the mesh honor the internal topography of the fault. When this internal layer reaches the surface of the mesh, some elements are squeezed and become significantly smaller, which can be seen in Fig. 4.11 as the thin dark red stripe of elements, which are 4–8 times smaller than the largest interior elements.

We ran performance experiments on Cray XK7 nodes, which are equipped with a single 16-core AMD Opteron 6272 2.1 GHz Interlagos CPU and a single K20X GPU. From Fig. 4.12 we see that the LTS-CPU version is 3.9x faster (in runtime) as compared to the non-LTS CPU version, which shows an LTS efficiency of 95%. The LTS GPU version managed an additional 7.4x speedup, yielding 28.9x total speedup against the original CPU version. The high CPU and GPU LTS efficiency indicates that our partitioning solution is working very well for this real-world application example. To put these speedup numbers in perspective, using 40 nodes, the non-LTS CPU version required more than 9 min to finish the simulation, whereas the LTS GPU version can finish it in less than 20 s.

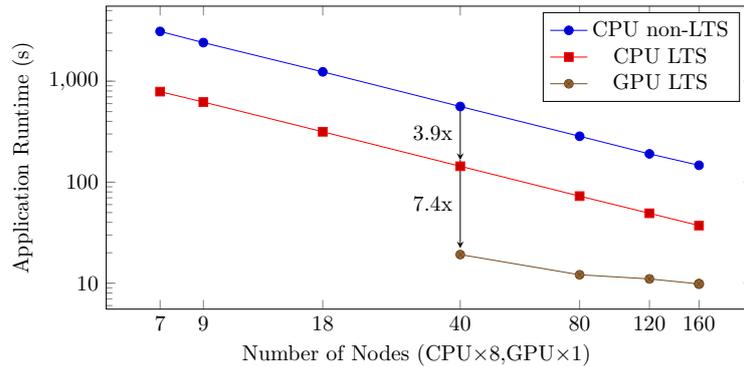


Figure 4.12. Runtime scaling (in seconds) comparing reference (non-LTS), LTS CPU, and LTS GPU versions running on the Tohoku mesh of 7.5M elements. The memory constraints limited the GPU version to start at 40 nodes. Experiments used only half the number of cores per node due to the shared floating point module architecture of the AMD Interlagos CPUs.

### 4.3.7 Additional partitioning options

The multiconstraint partitioning solutions presented previously are designed for the general case, where refined elements may be anywhere in the mesh. However, we investigated another solution we call *p-level-isolation* partitioning, which uses a similar approach to that used by Godel et al. [2010]. We hoped to replicate their results without manually altering the dual-graph representation by hand before partitioning. Instead, the PaToH library allows elements (cells) to be preassigned to a partition and further, for cells to be weighted by  $p$ -level. In theory, this allows us, for appropriate meshes, to partition the mesh such that only 1 partition contains elements with  $p > 1$ . This approach, however, puts a hard limit on strong scalability; At some point the partition with  $p > 1$  will not be able to shrink any further creating load imbalance if scaling continues. We present an example in Fig. 4.13, where the refinement in the center is fixed to partition 0. Unfortunately, PaToH is not able to maintain the load balance between the partitions when scaling the partition count despite being below the partition count limit. As the table shows, the imbalance grows quickly, even for low processor counts, as computed from the 300,000 element mesh example depicted in this figure.

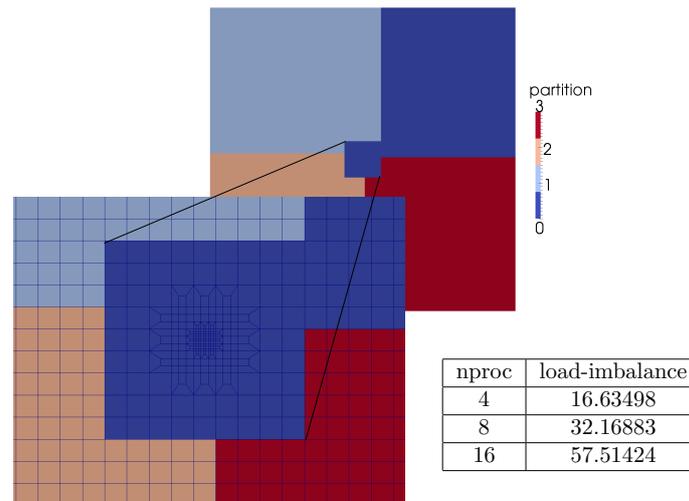


Figure 4.13. Isolation partitioning for test mesh with a 3-level refinement in the center.

## 4.4 Conclusion

In the field of seismic wave propagation, SEMs have seen great success for their impressive performance and flexibility utilizing user-defined hexahedral meshes. As mentioned, explicit time-stepping schemes enable codes such as SPEC3D to achieve impressive scaling efficiency for large problems on large CPU and GPU clusters. However, as we have seen in the previous chapter, the CFL stability criteria can significantly reduce overall performance.

In order to sidestep this CFL bottleneck, we introduced an LTS-Newmark method, which has been efficiently implemented in SPEC3D, adapted to the continuous nature of the higher-order polynomial basis functions. However, the multilevel nature of our LTS algorithm presents a load-balancing problem for the standard partitioning approach used by the reference code. By formulating this as a multiconstraint partitioning problem, we are able to test several competing partitioning strategies. The multiconstraint hypergraph partitioner provided by the PaToH library produces excellent results and, as noted, can effectively balance the communication and load-balancing constraints, where a similar approach provided by the MeTiS library is unable to compete.

We also propose the relatively simple solution presented as SCOTCH-P, a manual partition of each refinement  $p$ -level, that provides the best application performance for all of the meshes tested. Overall, we can conclude that LTS can be implemented efficiently for large-scale wave propagation. Additionally,

the GPU implementation demonstrates that combining algorithmic and architectural improvements can yield impressive speedups over the original CPU code. Given the move to GPU computing by many supercomputing centers, the effectiveness of the LTS-enabled GPU version should enable application scientists to pursue larger and more complex problems while maintaining or reducing time-to-solution.



## **Chapter 5**

# **Conclusions and Outlook**

## 5.1 Summary

This thesis presents the technologies to simulate wave propagation at the current cutting edge of performance. It presents these advancements in three parts:

1. The development and optimization of a GPU-capable seismic wave-propagation and tomography software package (SPECFEM3D).
2. The development and efficient implementation of the multilevel LTS-Newmark scheme to avoid time-stepping stability bottlenecks in finite-element meshes with localized refinement.
3. Ensuring load-balance (and efficiency) on many-node supercomputing systems for multilevel LTS algorithms via multiconstrained partitioning.

The final advancement of the LTS-aware partitioning schemes presented in the previous chapter presents the culmination of the research effort detailed in this thesis. Starting with Ch. 2, we present our development of a GPU-ready simulation package for the SEM used in SPECFEM3D. This chapter details optimizations for the SEM's algorithmic profile including several optimizations such as shared memory cache for intermediate variables, shared-node race-condition avoidance, and overlapped GPU-CPU transfers and MPI communications, all of which contribute to excellent single-GPU and many-GPU performance and scalability.

Beyond these traditional computational optimizations, we highlight the threaded I/O routines developed for tomography applications. Without overlapped and improved file writing and reading operations, the speedup achieved by the standard GPU optimizations is completely overshadowed by disk latency problems. With all these optimizations together, the simulation package SPECFEM3D runs 5–7x faster than comparably equipped CPU-only supercomputing nodes (even when scaled to many nodes).

Building on these performance improvements that take advantage of architectural advances, our second major contribution improves the time-stepping algorithms underlying the simulation itself. Detailed in Ch. 3, we develop a multi-level explicit LTS-Newmark scheme which can effectively mitigate CFL-stability bottlenecks. In a finite element mesh with regions of smaller elements, any standard explicit time-stepping scheme will be forced to take a time step  $\Delta t$  matched to the smallest elements, such that many elements take steps much smaller than optimal, which drastically reduces efficiency and increases wall-clock time required for a fixed amount of simulated time.

We avoid this problem by locally matching the time-step size  $\Delta t$  to the element size, which is grouped in what we have called  $p$ -levels. To ensure that our LTS algorithm is correct, the LTS chapter provides theoretical accuracy results in addition to numerical accuracy and stability experiments, validating the correctness of our multilevel scheme. The algorithmic results extend on previous two-level methods to work on an arbitrary number of time-stepping levels, including notes on how to adapt the scheme to work with the first-order absorbing boundary conditions. These derivations also contain algorithmic details on how our scheme has been implemented for the continuous basis functions used by the SEM in SPECFEM3D, which has been previously avoided by utilizing DG finite-element schemes. However, SEMs remain very popular and our algorithmic advancements enable LTS for them as well, at very high efficiency. By avoiding unnecessary operations, the LTS extension in SPECFEM3D can reach the mesh-dependent theoretical LTS speedup for both CPUs and GPUs at more than 90% efficiency on a single CPU or GPU.

In order to consider the presented simulation package ready for practical applications, it must work well, not only on a single node, but also on many-node supercomputing clusters. The recursive nature of the multilevel LTS algorithm presented in Ch. 3 makes parallelization difficult due to the uneven progression of steps across elements, creating a load-balancing problem when the simulation is parallelized using a standard mesh-partitioning strategy. Presented in Ch. 4, we utilize multiconstraint partitioning strategies to solve this problem for multilevel LTS schemes, with an implementation for our LTS-Newmark extension in SPECFEM3D. The best-performing partitioning strategy, dubbed *SCOTCH-P*, partitions each refinement level ( $p$ -level) separately, using a simple greedy recombination strategy that turns out to work very well in practice. We also considered the more elegant solution using the multiconstraint hypergraph partitioning library PaToH, which provides a similar solution to *SCOTCH-P*, albeit in a single step. This hypergraph solution, however, falls slightly short in scalability tests and is thus rejected as the default strategy unless improvements can be made to the PaToH for our LTS use case.

This LTS load-balancing chapter already summarizes our results, as it is the natural culmination of this research work, but the results deserve to be mentioned again. Unlike the GPU speedup results from Ch. 2, we cannot present a single performance speedup for LTS, as it depends on the finite element mesh in question. The Tohoku fault application example presented had a modest theoretical speedup of only about 4x, where we also considered several meshes with speedups ranging from 2x–22x. In fact, Sec. 3.7.4 shows that the implementation works (efficiently) for a mesh with a theoretical speedup of up to 100x.

With each chapter's contribution reviewed, we turn again to the presented application example from the introduction. This allows us to make the contribution of this thesis concrete, especially for the people who may utilize these contributions for a real application.

## 5.2 Revisited: Application Example

To recall, we set up a domain covering Switzerland in order to simulate an event in the eastern Alps on the French border. We considered recordings from two stations, SENIN close by at 90 km and FUORN in the far eastern part at 270 km distance. We additionally added a localized refinement surrounding the source with elements that were up to 32 times smaller than the majority of (nearly) uniform bulk mesh elements, which can be seen in Fig. 5.1. Without this refinement, a single GPU could simulate almost 200 s of wave propagation (simulated time) in less than 6 min of wall-clock time. With the refinement (only 3% of elements), this became nearly 2.5 hr. Of course, this drastic increase in wall-clock time provides a clear motivation to develop (and implement) the LTS scheme to allow the time steps to fit the element size, minimizing the additional work created by the refinement.

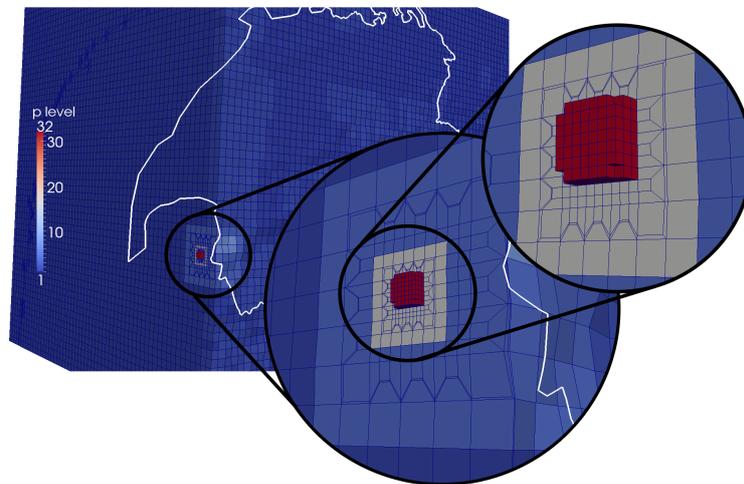


Figure 5.1. Switzerland example with localized refinement at source.  $\Delta t_{\min} = \Delta t_{\max}/32$ .

The mesh and its distribution of elements (with source refinement) can be seen in Fig. 5.2. Using the theoretical speedup formula (3.46) from Sec. 3.4.1,

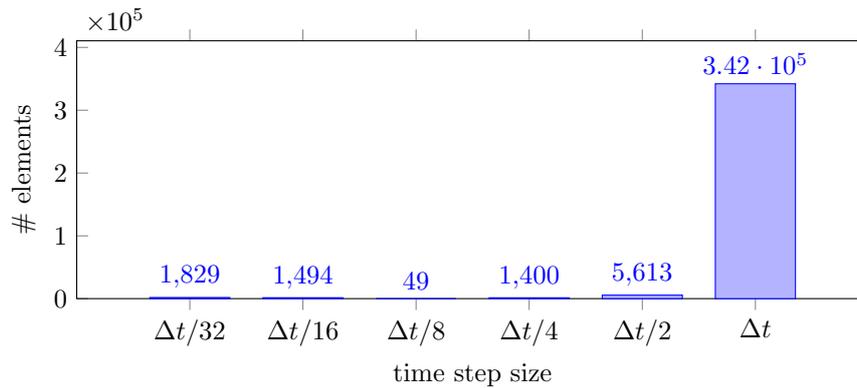


Figure 5.2. Distribution of element sizes according to time step size for the mesh of Switzerland with a localized refinement at the earthquake source.

the following calculation yields the theoretical speedup for this mesh:

$$\begin{aligned}
 \text{theoretical speedup} &= (32 \times 352482) / (32 \times 1829 \\
 &\quad + 16 \times 1494 \\
 &\quad + 8 \times 49 \\
 &\quad + 4 \times 1400 \\
 &\quad + 2 \times 5613 \\
 &\quad + 342097) = 25.5.
 \end{aligned}$$

As noted previously, this simplistic set of refined elements is meant to model existing or future scenarios. Fault modeling like in the Tohoku example in Ch. 4 already benefits by a factor of 4, and one can imagine crustal modeling applications to embed water or sediment layers at the surface. Minisini et al. [2013] demonstrated a speedup of 4x with a two-level scheme applied to embedded salt bodies and internal topography within a tetrahedral mesh. A move to our multilevel scheme, which allows more elements to take steps closer to their optimal step size, would likely increase this speedup by 1.5–2x. They additionally note that they were unable to run in parallel on a multinode system due to the LTS load-balancing problem when running in parallel using MPI — something this thesis has shown can be solved efficiently.

With the motivations clear, let us examine how well our full-stack solution performs on the Switzerland application example. As noted many times, the ability of all versions of the code to run in parallel across many nodes is critical for both runtime and memory concerns. In fact, due to the increased memory usage from our LTS implementation, the GPU version cannot run on a single

node at all. We run scaling experiments (Fig. 5.3) on the hybrid CPU/GPU cluster *Piz Daint* located at the CSCS. Each node is equipped with a single Intel 8-core CPU and a single NVIDIA K20X Tesla GPU. The 350,000 elements of the mesh make it fairly small, but its resolution is more than sufficient given the scale of velocity model features commonly seen in more recent tomographic models. Given its size, we only scale up to 16 nodes, that is 128 CPU cores or 16 GPUs.

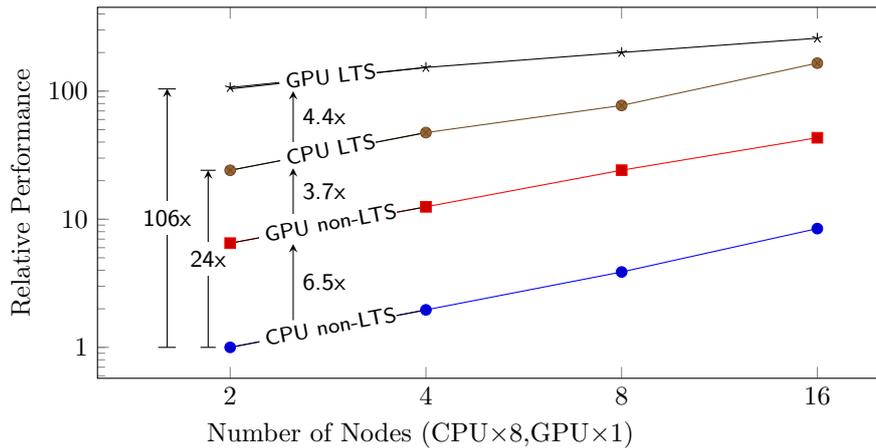


Figure 5.3. Relative (vs. non-LTS CPU version) performance of non-LTS and LTS versions of CPU and GPU code.

We summarize these experiments in Fig. 5.3 where we present the (relative) performance of SPECFEM3D running on CPUs and GPUs, using both the original non-LTS–Newmark scheme and our newly developed LTS–Newmark scheme. Note in particular that the scaling is plotted on a logarithmic scale, as the very large speedup (106x) between the non-LTS CPU version and the LTS GPU version would otherwise make the other curves impossible to distinguish. Each curve represents the performance ([simulated time/second]) relative to the non-LTS CPU version at 2 nodes (16 processes). Thus we see that the non-LTS GPU version is 6.5x faster (node-to-node) and the CPU LTS version is 24x faster. The GPU LTS version is 104x faster, which amounts to 75% of the theoretical LTS speedup. This reduced efficiency for the GPU version is a known effect of the additional LTS overhead and provides future opportunities for future optimizations. This efficiency generally increases when the mesh has a smaller LTS speedup, as seen in the example meshes from Ch. 4.

Despite some GPU inefficiencies, it nevertheless achieves more than 100x speedup over the reference CPU version without LTS, a truly transformative change for someone trying to run a large number of simulations using this mesh.

Of course, however, the greater point of this speedup is to allow modeling of fine-scale features without drastically impacting the cost of a simulation. These successful optimizations mean that we can model features that require element refinement without worrying that the simulation will require much more simulation time — by this metric our LTS solution on both CPUs and GPUs is very successful.

### 5.3 Final Discussion

The final results presented in this conclusion culminate several years of work that build constructively to produce a next-generation wave-propagation simulation package. However, the scope of this thesis leaves several open problems and topic areas still ripe for potential advancement. The strong scaling efficiency of the LTS-GPU version does leave room for improvement. The partitioning strategy and implementation for the GPU leaves very few elements per GPU at the lowest time-stepping levels. The overhead to setup and launch GPU kernels for so few elements at the lowest levels ends up dominating the runtime, which is an unavoidable cost in our particular parallelization strategy.

The CPU version did not experience the overhead seen by the GPU version as strongly due to cache effects and lower function-call overhead. However, if the cache utilization of the CPU version is improved, which we predict will increase performance by 55% (see Sec. 2.6), it might be that the scaling performance of the CPU LTS version will also suffer. As opposed to splitting the levels evenly across partitions, we might split the work using a simpler single-constraint partition such as SCOTCH. However, instead of computing element updates one level at a time, we build a queue of (independent) element updates across all levels, such that each processor is kept busy with element updates. This concurrency technique, however, would represent a nontrivial deviation from the standard computational structure within a package such as SPECFEM3D, and would require the corresponding developer investment to implement.

From the roofline analysis in Ch. 2, one might expect that our GPU version of SPECFEM3D should be able to achieve closer to the maximum roofline performance (it gets 55% of the modeled peak performance given the arithmetic intensity ( $AI$ )). Here we again expect a small boost of 15% by improving the layout of the DOFs. Future GPUs with more sophisticated caches may yield more performance here.

We were careful to take advantage of all the well-known GPU features such as shared memory, coalesced memory transfers, and the optimized atomic oper-

ations pipeline. However, a more detailed analysis coupled with more specific modeling requirements may yield unseen improvements, in part to the problems thus faced regarding register pressure and spilling, due to the complexity of the elasticity model.

This thesis presented a full-stack solution to optimized wave propagation for large-scale problems on large-scale computing systems. We have demonstrated that it is possible to extend a nontrivial application codebase (such as SPEC-FEM3D) to work with the newest in computing architectures (such as GPUs) and to integrate new time-stepping schemes (such as LTS-Newmark) to drastically accelerate the code. Of course, these contributions came at a high cost, in terms of developer investment and knowledge. This investment is, additionally, not a one-off cost, but extends to the maintenance of this code, as the physical modeling becomes more sophisticated, with the addition of more physical parameters such as attenuation, gravity, and anisotropy, which have been entirely ignored in this thesis. It is, of course, possible that these can be easily integrated into the GPU and LTS versions of the code, but they are beyond the reach of a quick fix by someone without the numerical, software, and computing knowledge — in short they need a trained computational scientist helping to maintain the code. However, assuming that SPEC-FEM3D has a sufficiently large user base, and will continue to see applications such as the Tohoku earthquake example in Sec. 4.3.6 (28x speedup) or the Switzerland example (106x speedup), LTS combined with a GPU is enough of a performance gain to invest the extra development and maintenance effort.

# Bibliography

- Aykanat, C., Cambazoglu, B. B. and Uçar, B. [2008]. Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices, *Journal of Parallel and Distributed Computing* **68**(5): 609–625.
- Bader, M. and Zenger, C. [2006]. Cache oblivious matrix multiplication using an element ordering based on a Peano curve, *Linear Algebra and its Applications* **417**(2-3): 301–313.
- Beyreuther, M., Barsch, R., Krischer, L., Megies, T., Behr, Y. and Wassermann, J. [2010]. ObsPy: A Python Toolbox for Seismology, *Seismological Research Letters* **81**(3): 530–533.
- Canuto, C., Hussaini, M. Y., Quarteroni, A. and Zang, T. A. [2006]. *Spectral Methods - Fundamentals in Single Domains*, Springer.
- Çatalyürek, U. V. and Aykanat, C. [1999]. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*, Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at <http://bmi.osu.edu/~umit/software.htm>.
- Cecka, C., Lew, A. J. and Darve, E. [2011]. Assembly of finite element methods on graphics processors, *International Journal for Numerical Methods in Engineering* **85**(5): 640–669.
- Chevalier, C. and Pellegrini, F. [2008]. PT-Scotch: A tool for efficient parallel graph ordering, *Parallel Computing* **34**(6-8): 318–331.
- Clayton, R. and Engquist, B. [1977]. Absorbing boundary conditions for acoustic and elastic wave equations, *Bulletin of the Seismological Society of America* **67**(6): 1529–1540.

- Diaz, J. and Grote, M. J. [2009]. Energy Conserving Explicit Local Time Stepping for Second-Order Wave Equations, *SIAM Journal on Scientific Computing* **31**(3): 1985–2014.
- Diehl, T., Kissling, E., Husen, S. and Aldersons, F. [2009]. Consistent phase picking for regional tomography models: application to the greater Alpine region, *Geophysical Journal International* **176**(2): 542–554.
- Droux, J. [1993]. An algorithm to optimally color a mesh, *Computer methods in applied mechanics and engineering* **104**: 249–260.
- Dumbser, M. and Käser, M. [2006]. An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes - II. The three-dimensional isotropic case, *Geophysical Journal International* **167**(1): 319–336.
- Dumbser, M., Käser, M. and Toro, E. F. [2007]. An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes - V. Local time stepping and p-adaptivity, *Geophysical Journal International* **171**(2): 695–717.
- Engquist, B. and Majda, A. [1977]. Absorbing boundary conditions for numerical simulation of waves, *Proceedings of the National Academy of Sciences* **31**(139): 629–651.
- Fichtner, A., Kennett, B. L. N., Igel, H. and Bunge, H.-P. [2009]. Full seismic waveform tomography for upper-mantle structure in the Australasian region using adjoint methods, *Geophysical Journal International* **179**(3): 1703–1725.
- Galvez, P., Nissen-Meyer, T., Ampuero, P. and Luis, A. D. [2010]. 3D rupture dynamics with unstructured spectral elements and a flux-based fault solver, *ESD 2010* p. 29.
- Godel, N., Schomann, S., Warburton, T. and Clemens, M. [2010]. GPU accelerated Adams-Bashforth multirate discontinuous Galerkin FEM simulation of high-frequency electromagnetic fields, *IEEE Transactions on Magnetics* **46**(8): 2735–2738.
- Gottlieb, S. [2005]. On High Order Strong Stability Preserving Runge–Kutta and Multi Step Time Discretizations, *Journal of Scientific Computing* **25**(1): 105–128.

- Grote, M. J., Mehlin, M. and Mitkova, T. [2014]. Runge-Kutta Based Explicit Local Time-Stepping Methods for Wave Propagation, *Technical report*, University of Basel.
- Grote, M. J. and Mitkova, T. [2010]. Explicit local time-stepping methods for Maxwell's equations, *Journal of Computational and Applied Mathematics* **234**(12): 3283–3302.
- Grote, M. J. and Mitkova, T. [2013]. High-order explicit local time-stepping methods for damped wave equations, *Journal of Computational and Applied Mathematics* **239**(1): 270–289.
- Heinecke, A., Breuer, A., Rettenberger, S., Bader, M., Gabriel, A.-A., Pelties, C., Bode, A., Barth, W., Liao, X.-K., Vaidyanathan, K., Smelyanskiy, M. and Dubey, P. [2014]. Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers, pp. 3–14.
- Hesthaven, J. and Warburton, T. [2008]. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, Springer-Verlag New York.
- Holk, E., Pathirage, M., Chauhan, A., Lumsdaine, A. and Matsakis, N. D. [2013]. GPU Programming in Rust: Implementing High-Level Abstractions in a Systems-Level Language, *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, IEEE, pp. 315–324.
- Karypis, G. and Kumar, V. [1998]. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, *SIAM Journal on Scientific Computing* **20**(1): 359–392.
- Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P. and Fasih, A. [2012]. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation, *Parallel Computing* **38**(3): 157–174.
- Klöckner, A., Warburton, T., Bridge, J. and Hesthaven, J. [2009]. Nodal discontinuous Galerkin methods on graphics processors, *Journal of Computational Physics* **228**(21): 7863–7882.
- Komatitsch, D., Erlebacher, G., Göldeke, D. and Michéa, D. [2010]. High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster, *Journal of Computational Physics* **024**: 1–28.

- Komatitsch, D., Michéa, D. and Erlebacher, G. [2009]. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA, *Journal of Parallel and Distributed Computing* **69**(5): 451–460.
- Komatitsch, D. and Tromp, J. [1999]. Introduction to the spectral element method for three-dimensional seismic wave propagation, *Geophysical Journal International* **139**(3): 806–822.
- Komatitsch, D. and Tromp, J. [2002]. Spectral-element simulations of global seismic wave propagation-I. Validation, *Geophysical Journal International* **149**(2): 390–412.
- Komatitsch, D. and Tromp, J. [2003]. A perfectly matched layer absorbing boundary condition for the second-order seismic wave equation, *Geophysical Journal International* **154**(1): 146–153.
- Komatitsch, D., Tsuboi, S. and Ji, C. [2003]. A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the Earth Simulator, *Proceedings of the 2003 ACM/IEEE conference on Supercomputing* .
- Krenk, S. [2006]. Energy conservation in Newmark based time integration algorithms, *Computer Methods in Applied Mechanics and Engineering* **195**(44-47): 6110–6124.
- Krivelevich, M. [2002]. Coloring random graphs—an algorithmic perspective, *Proc. 2nd Colloquium on Mathematics and Computer Science*, pp. 175–195.
- Madec, R., Komatitsch, D. and Diaz, J. [2009]. Energy-conserving local time stepping based on high-order finite elements for seismic wave propagation across a fluid-solid interface, *Computer Modeling in Engineering and Sciences* **49**(2): 163–189.
- Markall, G. G., Slemmer, A., Ham, D. D., Kelly, P. P., Cantwell, C. and Sherwin, S. [2013]. Finite element assembly strategies on multi-and many-core architectures, *International Journal for Numerical Methods in Fluids* **71**(1): 80–97.
- Minisini, S., Zhebel, E., Kononov, A. and Mulder, W. A. [2013]. Local time stepping with the discontinuous Galerkin method for wave propagation in 3D heterogeneous media, *Geophysics* **78**(3): T67–T77.
- Nissen-Meyer, T., Fournier, A. and Dahlen, F. a. [2008]. A 2-D spectral-element method for computing spherical-earth seismograms-II. Waves in solid-fluid media, *Geophysical Journal International* **174**(3): 873–888.

- Peter, D., Komatitsch, D., Luo, Y., Martin, R., Le Goff, N., Casarotti, E., Le Locher, P., Magnoni, F., Liu, Q., Blitz, C., Nissen-Meyer, T., Basini, P. and Tromp, J. [2011]. Forward and adjoint simulations of seismic wave propagation on fully unstructured hexahedral meshes, *Geophysical Journal International* **186**(2): 721–739.
- Peter, D., Tape, C., Boschi, L. and Woodhouse, J. H. [2007]. Surface wave tomography: global membrane waves and adjoint methods, *Geophysical Journal International* **171**(3): 1098–1117.
- Pharr, M. and Mark, W. [2012]. ispc: A SPMD compiler for high-performance CPU programming, *Innovative Parallel Computing (InPar)* .
- Pulido, N., Ojeda, A., Atakan, K. and Kubo, T. [2004]. Strong ground motion estimation in the Sea of Marmara region (Turkey) based on a scenario earthquake, *Tectonophysics* **391**(1-4): 357–374.
- Rietmann, M., Grote, M. J., Peter, D., Schenk, O. and Ucar, B. [2015]. Load-Balanced Local Time Stepping for Large-Scale Wave Propagation, *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium, IPDPS'15*, IEEE Computer Society. (acceptance rate: 21.8%, 108/496).
- Rietmann, M., Messmer, P., Nissen-Meyer, T., Peter, D., Basini, P., Komatitsch, D., Schenk, O., Tromp, J., Boschi, L. and Giardini, D. [2012]. Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC'12*, IEEE Computer Society Press. (acceptance rate: 21%, 100/472), pp. 38:1–38:11.
- Rietmann, M., Peter, D., Grote, M. J. and Schenk, O. [2014]. High Performance Newmark Load Time Stepping with Applications in Large Scale Seismic Wave Propagation, *Technical report*, USI Lugano.
- Shapiro, N. M., Campillo, M., Stehly, L. and Ritzwoller, M. H. [2005]. High-resolution surface-wave tomography from ambient seismic noise., *Science (New York, NY)* **307**(5715): 1615–8.
- Shimokawabe, T., Aoki, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., Nukada, A. and Matsuoka, S. [2011]. Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis - SC '11* .

- Stacey, R. [1988]. Improved transparent boundary formulations for the elastic-wave equation, *Bulletin of the Seismological Society of America* **78**(6): 2089–2097.
- Tape, C., Liu, Q., Maggi, A. and Tromp, J. [2009]. Adjoint tomography of the southern California crust., *Science (New York, NY)* **325**(5943): 988–92.
- Tromp, J., Komattisch, D. and Liu, Q. [2008]. Spectral-element and adjoint methods in seismology, *Communications in Computational Physics* **3**(1): 1–32.
- Tromp, J., Luo, Y., Hanasoge, S. and Peter, D. [2010]. Noise cross-correlation sensitivity kernels, *Geophysical Journal International* **183**(2): 791–819.
- Tromp, J., Tape, C. and Liu, Q. [2004]. Seismic tomography, adjoint methods, time reversal and banana-doughnut kernels, *Geophysical Journal International* **160**(1): 195–216.
- Volkov, V. and Demmel, J. [2008]. Benchmarking GPUs to tune dense linear algebra, *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, number November, IEEE.
- Williams, S., Waterman, A. and Patterson, D. [2009]. Roofline: an insightful visual performance model for multicore architectures, *Communications of the ACM* pp. 65–76.
- Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M. and Moshovos, A. [2010]. Demystifying GPU microarchitecture through microbenchmarking, *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, IEEE, pp. 235–246.
- Wright, S. and Nocedal, J. [1999]. *Numerical Optimization*, Springer New York.
- Zampieri, E. and Tagliani, A. [1997]. Numerical approximation of elastic waves equations by implicit spectral methods, *Computer Methods in Applied Mechanics and Engineering* **144**(1-2): 33–50.
- Zhu, H., Bozdog, E., Peter, D. and Tromp, J. [2012]. Structure of the European upper mantle revealed by adjoint tomography, *Nature Geoscience* **5**(7): 493–498.