A SCALABLE DISTRIBUTED FRAMEWORK FOR PARALLEL—ADAPTIVE SPACETIME DISCONTINUOUS GALERKIN SOLVERS WITH APPLICATION TO MULTISCALE EARTHQUAKE SIMULATION

BY

AMIT MADHUKAR

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Mechanical Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Abstract

The causal Spacetime Discontinuous Galerkin (cSDG) method is a powerful numerical scheme for solving hyperbolic systems. cSDG solvers construct asynchronous, unstructured spacetime meshes that reflect the target system's characteristic structure. When combined with spacetime discontinuous Galerkin bases, causal spacetime meshes localize the solution process to small clusters of spacetime elements called *patches* such that a localized implicit solution on each patch depends only on adjacent previously-solved *predecessor elements* and prescribed initial–boundary data. Starting with a simplicial space mesh with all vertices assigned initial-time coordinates, cSDG solvers advance a space-like front mesh through the spacetime analysis domain. Subsequent front-mesh configurations — in the form of bumpy terrains with independent time coordinates for each vertex — separate the previously-solved and unsolved parts of the analysis domain. Discrete increments of a vertex time coordinates advance the front locally subject to a *causality constraint* that requires every front configuration to be space-like. A simplicial spacetime mesh covering the local region between two front configurations defines a new patch on which a local, implicit and unconditionally stable cSDG problem is immediately solved. The new configuration then replaces its predecessor as the current front. The simulation terminates when the entire front-mesh passes the final time of the simulation. The localized cSDG solution procedure delivers element-wise balance properties and linear computational complexity in the number of patches.

The cSDG method supports an extremely dynamic form of adaptive spacetime meshing capable of capturing fast-moving solution features such as sharp wavefronts and dynamic fracture. Adaptive meshing operations execute at the same patch-level granularity as the localized solution scheme. If a newly-solved patch fails its adaptive error tolerances, we discard the patch solution, refine the current front mesh below the failed patch, and restart the spacetime meshing process to generate simultaneous local refinement in both space and time. Other adaptive operations, including coarsening, edge flips, and vertex motion (for mesh smoothing, interface tracking, *etc.*) are implemented by special patch configurations that perform the desired operation continuously in spacetime with no need for accuracy-limiting solution projections. The elimination of global time-step constraints, as in conventional time marching schemes, makes cSDG adaptive meshing particularly powerful.

The cSDG method possesses a rich parallel structure in which the local operations — spacetime patch construction, patch solution, and adaptive meshing — form an embarrassingly parallel unit of computation. Only procedures required to prevent interference between patches involved in simultaneous parallel processing break the embarrassingly parallel structure. We show that standard parallel frameworks for PDE solvers are either incompatible with or would seriously degrade the strengths of cSDG solvers. In particular, the Bulk Synchronous Parallel (BSP) model (as in standard time marching) is incompatible with the cSDG method's asynchronous, fine-grained structure, and the Domain Decomposition Method (DDM) for parallelization requires expensive rebalancing computations that cannot keep pace with fine-grained cSDG adaptive meshing. We abandon the BSP model and the DDM and describe a task-based asynchronous framework for distributed parallel–adaptive cSDG solvers that is deployable on distributed-memory clusters and that matches the cSDG method's structure. It features patch-based parallelization, a latency tolerant scheme for accessing global front-mesh data (the only distributed data structure), and asynchronous probabilistic load and data balancing methods.

We demonstrate the effectiveness of the parallel–adaptive cSDG method in multi-scale earthquake simulations, an application in which fault systems extending for hundreds of kilometers may require millimeter resolution to capture fine-scale response within rupture process zones. State-of-the art seismic simulation codes are unable to bridge the resulting extreme ranges of spatio-temporal scales, even on today's largest supercomputers. We combine a dynamic contact model and a slip-weakening friction model with a cSDG model for linear elastodynamics to build a dynamic rupture model for fault dynamics. We verify our model with dynamic-rupture benchmark problems from the Southern California Earthquake Center (SCEC) and demonstrate the power of high-order cSDG models and adaptive spacetime meshing in bridging the broad range of spatio-temporal scales in these problems. While our results are in broad agreement with previous solutions, we uncover details of high-frequency response that are missing in previous solutions. We also discover a new dipole-like feature at fault branch points that can lead to separation under certain circumstances.

To my father and mother

Acknowledgments

I would like to express my gratitude to Professor Robert Haber who served as my advisor. I am indebted to him for his insight and guidance which has proven to be invaluable during my PhD research. I would also like to acknowledge Professors Paul Fischer, Ahmed Elbanna, and Nikhil Chandra Admal for graciously agreeing to serve on my committee.

I am also thankful to Professor Reza Abedi for taking the time to guide me through the intricacies of the cSDG code, to Professor Volodymyr Kindratenko for providing the expertise and computational resources without which none of this would have been possible, and to Professor Ahmed Elbanna for stimulating and rewarding discussions throughout the development of the seismic code. I would also like to thank my fellow graduate students, Christian J. Howard, and Xiao Ma whose help and encouragement was greatly appreciated.

I would not have come this far were it not for the constant love and support of my family, for which I am deeply grateful.

Last, but by no means the least, I gratefully acknowledge financial support from the College of Engineering through the Carver Fellowship and the Graduate College through the Illinois Distinguished Fellowship program. The research in this dissertation was funded in large part by the National Science Foundation under the SPX Grant No. 1725544 and by the National Center for Supercomputing Applications at the University of Illinois Urbana–Champaign.

Table of contents

Chapter 1

Introduction

This dissertation describes the development of a novel distributed parallel–adaptive implementation of the causal Spacetime Discontinuous Galerkin (cSDG) method and its application to extreme multiscale problems arising in seismology. The underlying serial cSDG method for hyperbolic problems [1] features solutions on unstructured spacetime grids that satisfy a causality constraint that localizes both adaptive mesh generation and the discontinuous Galerkin solution procedure. This localization enables and supports element-wise balance properties, element durations governed exclusively by local wave speeds and mesh geometry (versus globally uniform time steps in traditional synchronous solvers), a highly dynamic form of adaptive spacetime meshing, and a high-level parallel structure that encompasses both the localized meshing and solution procedures. In combination with a dynamic contact algorithm based on weak enforcement of Riemann solutions [2], the serial cSDG method has proven highly effective in modeling complex multiscale dynamic fracture [3] and shows great promise for modelling earthquake ruptures. However, modifications to the cSDG formulation and its adaptive meshing strategy are needed to accommodate and improve fault physics models, and a complete reworking of existing frameworks for parallel finite element solutions is required to realize the method's full potential for parallel scaling on exascale platforms.

A main contribution of this dissertation is the development of an asynchronous, task-based, and barrier-free parallel–adaptive cSDG solver for hyperbolic partial differential equations (PDEs). We replace the standard Bulk Synchronous Parallel model [4] (time marching) with the asynchronous solution scheme of the cSDG method and thereby eliminate the synchronization barriers that plague many other parallel codes. The parallel architecture uses causal spacetime patches to define a common granularity for parallel execution of adaptive meshing, localized finite element solutions, and dynamic balancing of data and computational load. We implement asynchronous process classes that (i) serve distributed global mesh data, (ii) gather/scatter coherent fragments of the global data, and (iii) perform embarrassingly parallel adaptive meshing and cSDG solutions over the fragments. We abandon the traditional Domain Decomposition Method [5, 6] and instead develop novel probabilistic methods to balance distributed data and computational load that can keep pace with dynamic cSDG adaptive meshing. We show that the resulting parallel scheme is scalable and latency-tolerant on a large distributed cluster.

A second focus of this dissertation is the application of the cSDG method to large, multiscale problems in earthquake-rupture dynamics. We adapt the Riemann contact solutions in [3] to implement a dynamic fault-response model that accurately captures the evolution of slip along fault interfaces. We develop specialized adaptive error indicators for earthquake rupture simulations to govern the cSDG method's highly dynamic and fine-grained mesh adaptivity. This allows us to overcome the multi-scale challenges of earthquake simulation problems where existing non-adaptive methods struggle to resolve critical features, such as rupture process zones and stress dipoles at branch-fault intersections. We can also model off-fault damage zones as networks of fine-scale faults in lieu of smeared elastoplastic models. Fault nucleation, extension, and coalescence are accurately captured to model off-fault damage accumulation.

The remainder of this dissertation is organized as follows. In Chapter 2 we provide the motivation and background information of the overall cSDG method. We describe the causal meshing procedure that is central to the method and briefly introduce the finite element formulation. We introduce the parallel–adaptive architecture in Chapter 3 along with the novel distributed algorithms developed to support the code. Optimizations and numerical results are presented to highlight the scalability of the resulting parallel architecture. Finally in Chapter 4, we develop an extension of the cSDG method applicable to earthquake rupture simulations. We demonstrate the method's dynamic mesh adaptivity that is capable of resolving the multi-scale nature of such simulations. We present results that showcases the improved resolution possible with the method. Finally, we demonstrate the use of our parallel–adaptive method to dramatically accelerate the solution of complicated seismic problems.

Chapter 2

The Causal Spacetime Discontinuous Galerkin (cSDG) Method

2.1 Introduction

The Galerkin method is a general class of numerical methods for solving partial differential equations (PDEs) that includes finite element (FE) and spectral element methods among others. The Galerkin approximation to the analytical solution of a PDE is the unique best approximation to the exact solution in a given finite-dimensional trial space as measured with respect to a given error norm.

Reed and Hill introduced the first discontinuous Galerkin (DG) method as a means to solve neutron transport problems in 1973 [7]. Numerical analysts now use DG methods to solve ordinary differential equations as well as hyperbolic, parabolic, and elliptic PDEs. A key feature of DG methods is the use of trial solution spaces that admit piecewise-continuous functions to form Galerkin approximations. In the context of DG finite element methods, trial solution spaces admit functions that are continuous on element interiors but admit discontinuities across element boundaries. The DG method has many advantages over classical finite volume and finite element methods. Due to the fact that inter-element continuity is not required, DG methods can support non-conforming meshes, a capability that can simplifying adaptive mesh generation. Parallelization of DG models is more tractable than conforming models since solutions involve only face-to-face information exchanges. A review of DG methods can be found in the work by Cockburn *et al.* [8].

Although spacetime finite elements had been considered (and largely rejected) decades earlier, Hughes and Hulbert [9] renewed interest in the topic with a time-discontinuous DG method that featured conforming bases on unstructured meshes within spacetime slabs having uniform thickness, Δt, while admitting jumps across constant-time surfaces between slabs. Wiberg *et al.* [10] and Li *et al.* [11] introduced h–adaptive meshing in space, allowing non-conforming meshes across constant-time boundaries and facilitating adaptive refinement and coarsening between slabs. However, the conforming basis within each slab requires re-computation of the entire slab's solution whenever any refinement occurs inside the slab.

Richter [12] was among the first to introduce a casual Spacetime Discontinuous Galerkin (SDG) method[1] that used unstructured causal meshing with bases that admit discontinuities across all spacetime inter-element boundaries with no requirement to organize elements into slabs. Thus, the concept of global time steps is replaced by the durations (time increments) of individual spacetime elements. When compared to semi-discrete methods, the spacetime counterparts have considerable benefits. Foremost of which, spacetime methods can produce fully unstructured and non-conforming meshes in both space and time, allowing for the development of powerful adaptive schemes. Yin *et al.* [13] extended the SDG method by applying unstructured, fully-discontinuous space-time discretizations to linear elastodynamics. Abedi *et al.* [1, 14, 15] introduced further refinements that led to more recent versions of the causal Spacetime Discontinuous Galerkin (cSDG) method.

The cSDG method has been applied to a variety of application domains. Causal spacetime meshing algorithms were developed over time in the work by Erickson *et al.* [16], Üngör and Sheffer [17] and Abedi *et al.* [18]. These powerful h-adaptive, non-conforming spacetime meshing capabilities have been applied to a number of applications, including linear elastodynamics utilizing a one-field [15, 19] and a three-field formulation [20]; fracture and crack-growth problems [2, 3, 21, 22]; hydraulic fracturing [23, 24]; hyperbolic advection–diffusion problems [25]; the radiative transfer equation [26]; and electromagnetism [27].

The remainder of this chapter presents the causal spacetime meshing algorithm in Section 2.2 and the spacetime discontinuous Galerkin formulation for linear elastodynamics in Section 2.3. These components are the foundation of the cSDG methods described and applied later in this dissertation.

2.2 Spacetime Meshing

The cSDG method works on meshes constructed in *spacetime*. A four-dimensional spacetime mesh is required for a problem that involves evolving behavior in a three-dimensional spatial

[1]Richter referred to his method as an explicit finite element method. This is a misnomer in that the method is more accurately described as locally implicit and has unconditional stability typical of implicit methods. Richter's use of explicit might have been motivated by the fact that his method localizes the solution process similar to an explicit method.

domain (3 spatial dimensions $\times$ time), for example. This contrasts with traditional *semi-discrete* methods where spatial dimensions are discretized independently of time.

The spacetime meshing algorithm, known as *Tent Pitcher* (developed by [16, 17]), produces a fully unstructured, *simplicial* spacetime mesh where the time step in each spacetime element only depends on the properties of the local, underlying space mesh. By weakly enforcing the governing equations over each spacetime element the cSDG method eliminates the need for a time-marching procedure. Global time-marching schemes suffer from the following drawbacks - either the maximum possible time step is constrained by the worst quality element in the space mesh [2], or a large coupled system has to be solved at each time increment. By construction, the Tent Pitcher algorithm avoids both these issues. In fact, by computing solutions within groups of spacetime elements, or *patches*, the computational cost is linear in the number of spacetime elements.

2.2.1 Causality and the Progress Constraint

The characterizing feature of hyperbolic PDEs is the notion of *causality*. Disturbances in initial data of hyperbolic boundary value problems travel in a "wave-like" nature along *characteristics* of the equation with a bounded wave speed. The characteristics of a hyperbolic spacetime PDE represents the flow of information through space with time.

Points in spacetime are partially ordered[3] by this concept of causality. We say that a point P *depends* on another point Q if and only if changes to physical quantities (such as temperature, displacement, etc.) at point P influences Q. The *domain of influence* of P is the set of points that depend on P and the *domain of dependence* is the set of points that P depends on. If the governing equations are linear and the material properties are homogeneous and isotropic, as in the case considered here, the wave speed ω is a constant and the domains of influence and dependence are circular cones as depicted in Figure 2.1. We say that one spacetime element $\triangle$ depends on another spacetime element $\triangle'$ if any point $P \in \triangle$ depends on any point $Q \in \triangle'$, *i.e.,* if Q is in the domain of influence of P.

We say that a face, or *facet*, F of a spacetime element is *causal* if it separates the cone of influence from the cone of dependence at every point on F (see Figure 2.1). Stated another way, causality requires each facet to be faster (or closer to horizontal, in spacetime) than the maximum wave speed, *i.e.,* $||\nabla F|| \leq 1/\omega$. In general, the wave speed $\omega(P)$ at each point P in the domain defines a scalar field over the domain. If a facet is causal, information only flows

[2]Work by [28] allows larger time steps than dictated by the worst quality element in explicit time-stepping methods by adaptively taking multiple stabilizing steps in each time increment

[3]A partially ordered set is a set P with a binary relation $R \subset P \times P$ satisfying the properties of *reflexivity*, *antisymmetry* and *transitivity* in set theory.

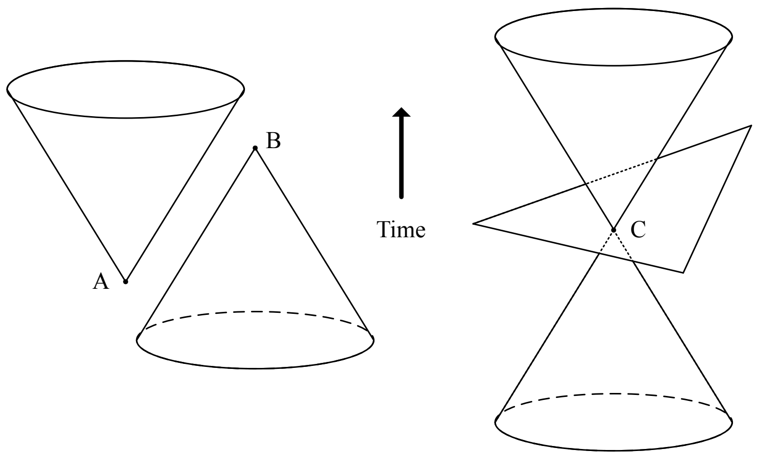


Figure 2.1: (Left) Points A and B are independent since B lies outside the cone of influence of A and A lies outside the cone of dependence of B. (Right) The facet centered on point C separates the cone of influence from the cone of dependence.

in one direction across that facet. The solution to the hyperbolic boundary value problem can be computed simultaneously in spacetime elements that obey the causality constraint and do not depend on each other.

Figure 2.2 illustrates the concept of causality for the simple case of 1D $\times$ time. The top part of the figure depicts an unstructured mesh generated by the Tent Pitcher algorithm where the arrows indicate the characteristics of the problem. In this case, characteristics propagate in both directions at a constant speed. The construction of element facets obeys the causality constraint so information flows only in one direction across each facet. We can thus label the earlier facets of an element as the *inflow* facets and the later ones as the *outflow* facets based on the direction of the flow of information. As long as the partial element ordering is observed, solutions to the boundary value problem can be computed locally and independently within causal elements. For example, if we consider all the level-1 elements, the solution within these elements only depends on the initial conditions along their bottom facets and boundary conditions along the left and right domain boundaries. As such, the solutions within the level-1 elements can be computed locally and concurrently. Any level-2 element can be solved as soon as its immediate level-1 neighbor has been solved, even if other level-1 element remains unsolved. It becomes clear from this simple example that a causal spacetime mesh enables asynchronous element-by-element solution with linear complexity.

On the other hand, the bottom part of Figure 2.2 depicts a noncausal mesh corresponding to a traditional time-marching scheme where each element depends on the other. In this case all elements must be solved together. Such a meshing scheme does not take advantage of the causal property inherent to hyperbolic PDEs.

To construct an efficient spacetime mesh, the Tent Pitcher algorithm groups elements

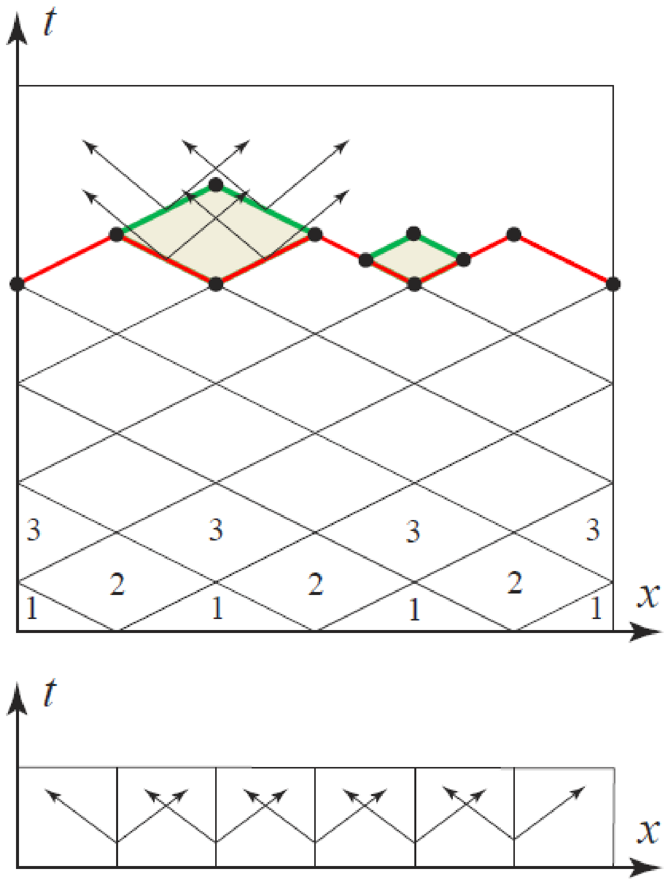


Figure 2.2: 1d × time domain meshed with causal (top) and noncausal (bottom) elements. Reproduced from [29].

in *patches* each containing a constant number of spacetime elements. All the elements in a patch must be solved simultaneously as facets of the elements within a patch are not causal. A patch can more formally be defined as the following:

Definition 2.1 (Spacetime patch Π)**.** A patch Π is a set of one or more spacetime elements belonging to a spacetime mesh Ω such that [30]:

1. for every element $A \in \Pi$, there exists $B \in \Pi$ such that A and B are coupled;
2. for every element $A \in \Pi$, if there exists $B \in \Omega$ such that A and B are coupled then $B \in \Pi$.

The spacetime mesh can be solved in a patch-by-patch manner that respects the partial ordering of points in the spacetime analysis domain. If we assume polynomial basis function of bounded degree the resulting system of equations in each patch has bounded size and can therefore be solved in constant time. We say that the solution strategy is *efficient* if the total computation time is proportional to the number of elements in the mesh.

In addition to the causality condition, it was determined that another so called *progress* constraint is needed to construct the spacetime mesh. Without this constraint, there may be cases where the construction of causal elements would make it impossible for subsequently constructed elements to obey the causality constraint, thus making progress impossible. Practically, this places a further limit on the maximum time-step that can be taken when constructing a new patch (see Section 2.2.2). We refer readers to the paper by [31] for a more detailed discussion on the causality and progress constraint.

2.2.2 Advancing Front Spacetime Meshing

The patch-by-patch solution strategy described in Section 2.2.1 respects the partial ordering of patches in spacetime. Our goal is to now incrementally construct the spacetime mesh that respects both the causality and progress constraint. This is achieved by the *advancing front* meshing procedure.

Definition 2.2 (Front τ)**.** For any time, the *front* τ is the graph of a continuous time function $\tau : \mathbb{E}^d \rightarrow \mathbb{R}$, such that within every triangle [31],

1. τ is linear;
2. $||\nabla\tau|| \leq 1/\omega$

i.e., the front is a *maximal* set of points such that no two points of τ influence each other.

The front is a d-dimensional piecewise linear terrain, a subset of the spacetime domain $\mathbb{E}^d \times \mathbb{R}$. Each point P on the front τ can be written as

$$P = (p, \tau(p)) \tag{2.1}$$

where p is the spatial projection of P and $\tau(p)$ is the temporal projection. When the current front τ is in context, we use the lower-case notation to represent vertex on that front, *i.e.,* $p \in \tau$. In the previous 1D $\times$ time case presented in Figure 2.2, the front is represented by the facets marked with the solid red line.

Given an initial triangulated front $\tau : \mathbb{E}^d \rightarrow \mathbb{R}$, Tent Pitcher selects an arbitrary local minimum vertex $P = (p, \tau(p))$ and moves it forward in time to a new point $P' = (p, \tau'(p))$, thus also advancing the local neighborhood. The volume between the new and old front is called a *tent* and the edge corresponding to the time increment, *i.e.,* PP', is called the *tent-pole*. The advancing front method is depicted for the case of 2d $\times$ time in Figure 2.3. The tent is decomposed into spacetime elements (or patch) sharing the noncausal edge PP' and thus, must be solved as a coupled system. However, since the causality constraint is placed on the maximum time step taken when advancing the front, each patch constructed by the advancing front method depends only on elements adjacent to its inflow boundary, in which the solution has already been computed. Therefore, the solution within the a new patch can be computed as soon as the patch is constructed. After solving a patch, we update the front by increasing the time value of the vertex pitched by the temporal span of the tent-pole, so the graph of the time function agrees with the new front. Equivalently, to update the front, we replace the inflow facets of the patch with the outflow facets of the patch.

The algorithm advances local minimum vertices since this guarantees a finite amount of progress can be made. A *local minimum* of the front τ is a vertex p such that $\tau(p) \leq \tau(q)$ for every vertex q that is a direct neighbor of p. A list of all local minima are maintained as a time ordered *heap*. When deciding the next vertex to pitch over, Tent Pitcher chooses the first entry in this heap, corresponding to the global minimum vertex. This heuristic approach to selecting the next pitchable vertex has been found to perform better than others.

The advancing front procedure thus allows us to *decouple* the patch generation process from the solution process. This feature has been used in the parallel implementation described in chapter 3 where the meshing and solution tasks have been separated into distinct units of work which can be carried out simultaneously.

Another unique capability of the advancing front algorithm is the inherent capability of localized mesh adaptivity operations. The solution within each patch produces an *a posteriori* estimate of the numerical error incurred during the solution. This error estimate can be used to drive an adaptive meshing algorithm as presented in [32]. Adaptive refinement and coarsening of the individual patches can occur locally and independently of each other where dictated by the error. Adaptive operations can be confined within the patch and the solution only needs to be recomputed within that region. Additionally, patch generation and adaptive operations can take place simultaneously with the SDG solution procedure in causal patches. This is an incredibly powerful feature of the SDG method that can be exploited by parallel execution.

2.3 Formulation of the Causal Spacetime Discontinuous Galerkin Method for Linear Elastodynamics

2.3.1 Spatial and Spacetime Analysis Domains

Consider an open *spatial domain*, $\omega \subset \mathbb{E}^d$, with a suitably regular boundary, $\partial\omega$. We seek a solution on the *spacetime domain*, $\Omega := \omega \times \mathcal{I} \subset \mathbb{E}^d \times \mathbb{R}$, in which $\mathcal{I} := (0, T)$ where 0 and

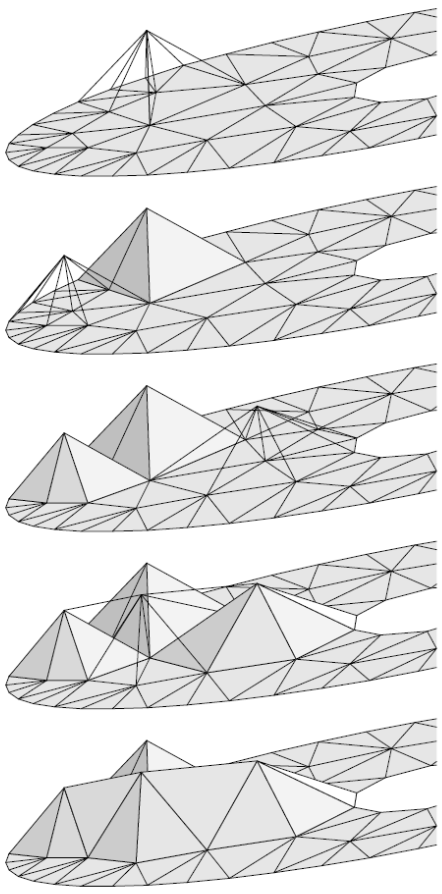


Figure 2.3: Pitching patches (or *tents*) in spacetime using the advancing front algorithm. Newly formed patches are shown in wireframe whereas solid surfaces are patches solved by the SDG solver. Note the partial ordering of patches in accordance to causality. The sequence is advancing from top to bottom while time is increasing upwards. Reproduced from [19]

T are the initial and terminal times for the simulation. We partition the boundary of Ω as $\partial\Omega = \overline{\partial\Omega_{\mathrm{I}} \cup \partial\Omega_{\mathrm{B}} \cup \partial\Omega_{\mathrm{T}}}$ in which $\partial\Omega_{\mathrm{I}}$ and $\partial\Omega_{\mathrm{B}}$ are open segments of $\partial\Omega$ on which initial and boundary conditions are enforced while no conditions are enforced on the terminal boundary, $\partial\Omega_{\mathrm{T}}$. We further partition $\partial\Omega_{\mathrm{B}}$ as $\partial\Omega_{\mathrm{B}} = \overline{\partial\Omega_{\mathrm{D}} \cup \partial\Omega_{\mathrm{N}}}$ in which $\partial\Omega_{\mathrm{D}}$ and $\partial\Omega_{\mathrm{N}}$ are the parts of $\partial\Omega_{\mathrm{B}}$ on which Dirichlet and Neumann boundary conditions are enforced.

2.3.2 Governing Equations and Initial/Boundary Conditions

The governing equations on Ω for linear elastodynamics, expressed in direct tensor notation, are

$$\mathbf{v} = \dot{\mathbf{u}}; \quad \mathbf{E} = \operatorname{sym} \nabla \mathbf{u} \tag{2.2a}$$

$$\dot{\mathbf{E}} = \operatorname{sym} \nabla \mathbf{v} \tag{2.2b}$$

$$\mathbf{S} = \mathbf{C}(\mathbf{E}); \quad \mathbf{p} = \rho \mathbf{g}(\mathbf{v}) \tag{2.2c}$$

$$\nabla \cdot \mathbf{S} + \rho \mathbf{b} = \dot{\mathbf{p}} \tag{2.2d}$$

in which $\mathbf{u}$, $\mathbf{v}$, $\mathbf{E}$, $\mathbf{S}$, $\mathbf{p}$, and $\mathbf{b}$ are the displacement, velocity, strain, stress, linear momentum density, and body force fields, while ρ and $\mathbf{C}$ are the mass-density and elasticity-tensor fields. The metric tensor, assuming a uniform Cartesian coordinate frame on ω for simplicity, is given by $\mathbf{g} := \delta_{ij} \mathbf{e}^i \otimes \mathbf{e}^j$. From here on we assume isotropic elastic response without loss of generality. Equations (2.2a) and (2.2b) are the kinematic compatibility relations. Equation (2.2b) is redundant if equation (2.2a) is strictly enforced, but is required if equation (2.2a) is only weakly enforced. Equation (2.2c) describes the linear constitutive relations, and (2.2d) is the Equation of Motion. We specify conforming initial displacement, velocity, and strain fields on $\partial\Omega_\mathrm{I}$: $\mathbf{u}_\mathrm{I}$, $\mathbf{v}_\mathrm{I}$, and $\mathbf{E}_\mathrm{I} := \operatorname{sym} \nabla \mathbf{u}_\mathrm{I}$ as well as prescribed velocity and traction fields on $\partial\Omega_\mathrm{D}$ and $\partial\Omega_\mathrm{N}$ to close the governing system.

We reduce system (2.2) to a single governing equation in $\mathbf{u}$ on $\Omega \setminus \Gamma$ as follows. We substitute $\mathbf{v}_\mathbf{u} := \dot{\mathbf{u}}$, $\mathbf{E}_\mathbf{u} := \operatorname{sym} \nabla \mathbf{u}$, $\mathbf{S}_\mathbf{u} := \mathbf{C}(\mathbf{E}_\mathbf{u})$, and $\mathbf{p}_\mathbf{u} := \rho \mathbf{g}(\dot{\mathbf{v}}_\mathbf{u})$ for, respectively, $\mathbf{v}$, $\mathbf{E}$, $\mathbf{S}$, and $\mathbf{p}$ in Systems (2.2) and (4.1) to strictly enforce eqs. (2.2a) and (2.2c). Strict enforcement of eqs. (2.2a) implies strict enforcement of eq. (2.2b). Thus, we obtain the Equation of Motion on Ω written in terms of $\mathbf{u}$,

$$\rho \mathbf{g}(\ddot{\mathbf{u}}) - \nabla \cdot [\mathbf{C}(\operatorname{sym} \nabla \mathbf{u})] = \rho \mathbf{b} \tag{2.3}$$

2.4 Patchwise cSDG Discretization

This section describes discrete approximations of continuum solutions to the initial–boundary value problem of linear elastodynamics by the cSDG method. In particular, we consider local approximations associated with causal patch meshes generated by procedures described in Section 2.3. Inter-element jump conditions and the Stokes Theorem are essential to the development, as with any discontinuous Galerkin formulation. However, spacetime element facets in patch meshes may have arbitrary orientations in $\mathbb{E}^d \times \mathbb{R}$, a space that lacks an inner product and, therefore, provides no objective means to construct normal vectors as required in tensorial representations of jump conditions and the Stokes Theorem.

Differential forms and the exterior calculus on manifolds provide an alternative mathematical framework that supports objective formulations of cSDG approximations that do not involve normal vectors. Out of necessity, we follow this approach below. However, our goal here is to convey to readers — including those unfamiliar with forms and exterior calculus — a high-level understanding of cSDG solution spaces and weak approximations. To this end, we present a reformulation of linear elastodynamics using forms and exterior calculus that omits low-level technical details.[4] For each equation in forms notation, we identify the corresponding tensorial equation in Subsection 2.3.2. Thus, for example, a reader can identify eqs. (2.2d) and (2.4d) below as the Equation of Motion expressed in alternative formats. This association is sufficient for our present purposes.

Jump conditions on spacetime interfaces emerge in the forms development as the jump parts of exterior derivative operators in the governing equations.[5] These include, but are not limited to, jump conditions on material-interface trajectories such the fault trajectories, Γ, introduced in Subsection 4.2.2. Due to their special orientations in $\mathbb{E}^d \times \mathbb{R}$, we can equip material-interface trajectories with purely spatial normal vectors in $\mathbb{E}^d$ that support tensorial formulations of jump conditions. In this special case, the jump conditions presented in tensor format in Subsection 4.2.2 are fully compatible with the jump conditions in the more general forms development.

2.4.1 Formulation of Linear Elastodynamics Using Differential Forms and Exterior Calculus on Manifolds

We replace the three kinematic fields in the tensorial formulation with differential forms as follows.[6] The vector displacement field, $\mathbf{u}$, is unchanged; *i.e.,* it is a 0-form. A differential 1-form with vector coefficients, $\boldsymbol{v}$, replaces the velocity field, $\mathbf{v}$, and a 1-form with symmetric second-order tensor coefficients, $\boldsymbol{E}$, replaces the strain field, $\mathbf{E}$. We combine $\boldsymbol{E}$ and $\boldsymbol{v}$ in a single spacetime 1-form, $\boldsymbol{\varepsilon} := \boldsymbol{E} + \boldsymbol{v}$, called the *strain–velocity.*

A d-form with symmetric second-order tensor coefficients, $\boldsymbol{S}$, replaces the stress tensor field, $\mathbf{S}$, and a d-form with vector coefficients, $\boldsymbol{p}$, replaces the linear-momentum-density vector field, $\mathbf{p}$. These combine to form a single d-form, $\boldsymbol{M} := \boldsymbol{p} - \boldsymbol{S}$, called the *spacetime momentum*

[4]A more detailed development can be found in [1, 20].

[5]Exterior derivatives of a BV field generally include an *absolutely continuous part* that holds in regions where the field is continuous, a *Cantor part* that we ignore in this work, and a *jump part* that acts across discontinuities. Typically, the exterior derivative operator, denoted as $\mathbf{d}$, contains all three parts. In this work, $\mathbf{d}$ denotes only the continuous part, and the jump part appears separately in the form of jump conditions.

[6]We denote differential forms in bold Italic font in contrast to tensors in bold upright font.

flux. The restriction of $\boldsymbol{M}$ to a d-manifold with arbitrary orientation in $\mathbb{E}^d \times \mathbb{R}$ delivers the flux of linear momentum across that manifold. A $(d+1)$-form with vector coefficients, $\boldsymbol{b}$, replaces the body-force vector field, $\mathbf{b}$.

Continuum solutions of hyperbolic problems generally include discontinuities in the response fields that arise from physical features such cracks, ruptures, and shocks. Discontinuous Galerkin approximations of hyperbolic solutions introduce additional discontinuities across element boundaries. The following reformulation of linear elastodynamics accommodates all such discontinuities across a general set of d-manifolds where jump conditions arise naturally as the jump parts of exterior derivative operators in the governing system.

We introduce the *jump set*, Γ^{J}, that contains all d-manifolds embedded in the spacetime analysis domain, Ω, across which jumps in the solution may occur for either physical or numerical reasons. We rewrite System (2.2) using differential forms as follows. The absolutely continuous part of the governing differential equations on $\Omega \setminus \Gamma^{\mathrm{J}}$ are written as

$$\boldsymbol{\varepsilon} = \operatorname{sym} \mathbf{d}\mathbf{u} \tag{2.4a}$$

$$\mathbf{d}\boldsymbol{\varepsilon} = \mathbf{0} \tag{2.4b}$$

$$\boldsymbol{M} = \mathcal{C}\left(\boldsymbol{\varepsilon}\right) \tag{2.4c}$$

$$\mathbf{d}\boldsymbol{M} = \rho \boldsymbol{b} \tag{2.4d}$$

in which $\mathbf{d}$ is the exterior derivative operator, the sym operator acts only on the spatial gradient part of $\mathbf{du}$, and $\mathcal{C}$ is a constitutive operator that maps strain-velocity 1-forms to spacetime-momentum-flux d-forms. The subequations in System (2.4) are the forms equivalents of their counterparts in System (2.2). Moreover, the solo eqs. (2.4a) and (2.4c) cover both branches of (2.2a) and (2.2c).

We add the jump parts of the exterior derivatives in System (2.4) to extend the governing equations from $\Omega \setminus \Gamma^{\mathrm{J}}$ to all of Ω. Thus, we have the jump conditions

$$\left(\overset{\star}{\mathbf{u}} - \mathbf{u}\right)^{\pm}\Big|_{\Gamma^{\mathrm{J}}} = \mathbf{0} \tag{2.5a}$$

$$\left(\overset{\star}{\boldsymbol{\varepsilon}} - \boldsymbol{\varepsilon}\right)^{\pm}\Big|_{\Gamma^{\mathrm{J}}} = \mathbf{0} \tag{2.5b}$$

$$\left(\overset{\star}{\boldsymbol{M}} - \boldsymbol{M}\right)^{\pm}\Bigg|_{\Gamma^{\mathrm{J}}} = \mathbf{0} \tag{2.5c}$$

in which a superscript $\pm$ denotes a trace on the $+$ or $-$ side of Γ^{J}, and a superposed $\star$ denotes a *target value* determined, depending on context, by a Riemann solution, initial or boundary condition, or a special physics model such as the fault-rupture model described in Subsection

4.2.3.

We obtain a single-field displacement formulation as follows. We substitute $\boldsymbol{\varepsilon}_{\mathbf{u}} := \operatorname{sym} \mathbf{d}\mathbf{u}$ for $\boldsymbol{\varepsilon}$ and $\boldsymbol{M}_{\mathbf{u}} := \mathcal{C}(\boldsymbol{\varepsilon}_{\mathbf{u}})$ for $\boldsymbol{M}$ in Systems (2.4) and (2.5) to strictly enforce eqs. (2.4a) and (2.4c). Strict enforcement of eq. (2.4a) implies strict enforcement of eq. (2.4b).[7] The governing system reduces to a second-order form of the Equation of Motion on $\Omega \setminus \Gamma^{\mathrm{J}}$ that is the forms version of eq. (2.3) (except now on $\Omega \setminus \Gamma^{\mathrm{J}}$).

$$\mathbf{d}\boldsymbol{M}_{\mathbf{u}} = \rho \boldsymbol{b} \tag{2.6}$$

Strict enforcement of eqs. (2.4a) and (2.4c) on $\Omega \setminus \Gamma^{\mathrm{J}}$ does not imply satisfaction or elimination of any jump conditions. Thus, we have

$$\left(\overset{\star}{\mathbf{u}} - \mathbf{u}\right)^{\pm}\Big|_{\Gamma^{\mathrm{J}}} = \mathbf{0} \tag{2.7a}$$

$$\left(\overset{\star}{\boldsymbol{\varepsilon}} - \boldsymbol{\varepsilon}_{\mathbf{u}}\right)^{\pm}\Big|_{\Gamma^{\mathrm{J}}} = \mathbf{0} \tag{2.7b}$$

$$\left(\overset{\star}{\boldsymbol{M}} - \boldsymbol{M}_{\mathbf{u}}\right)^{\pm}\Big|_{\Gamma^{\mathrm{J}}} = \mathbf{0} \tag{2.7c}$$

Compatible initial data, $\mathbf{u}_{\mathrm{I}}$, $\mathbf{v}_{\mathrm{I}}$, and $\mathbf{E}_{\mathrm{I}} := \operatorname{sym} \nabla \mathbf{u}_{\mathrm{I}}$, and Dirichlet and Neumann boundary data are specified as before.

2.4.2 Patch-wise cSDG Approximation on a Broken Sobolev Space

A typical cSDG patch mesh is comprised of a set of active spacetime elements on which the solution has not yet been computed and a set of previously-solved predecessor elements. We next formulate the cSDG method for computing an approximate solution on the active part of the patch. Let $\mathcal{P}$ denote the set of open spacetime element interiors in the current patch with the active subset, $\mathcal{P}_{\mathrm{a}} := \{\mathcal{Q} \in \mathcal{P} : \mathcal{Q} \text{ is active}\}$.

We seek patch-wise solutions that are smooth and continuous on all $\mathcal{Q} \in \mathcal{P}$ while admitting discontinuities across inter-element boundaries within $\mathcal{P}$ and between $\mathcal{P}$ and $\partial\Omega$. To this end, we construct a discrete subspace of a broken Sobolev space on $\overline{\mathcal{P}}$ to support the single-field cSDG discretization. Let $\mathscr{P}^k_{\mathcal{Q}}$ denote the space of polynomial functions of order k on $\mathcal{Q} \in \mathcal{P}_{\mathrm{a}}$. The solution space for displacement approximations of polynomial order k on any $\mathcal{Q} \in \mathcal{P}_{\mathrm{a}}$ is $\mathcal{U}^k_{\mathcal{Q}} := \left\{\mathbf{u} : u_i|_{\mathcal{Q}} \in \mathscr{P}^k_{\mathcal{Q}}\right\}$. We form a patch-wise displacement solution space of polynomial order k as a subspace of a broken Sobolev space on $\overline{\mathcal{P}_{\mathrm{a}}}$: $\mathcal{U}^k_{\mathcal{P}} := \left\{\mathbf{u} : \mathbf{u}|_{\mathcal{Q}} \in \mathcal{U}^k_{\mathcal{Q}} \ \forall\ \mathcal{Q} \in \mathcal{P}_{\mathrm{a}}\right\}$.

[7]We have $\mathbf{d}\boldsymbol{\varepsilon}_{\mathbf{u}} = \mathbf{d}(\operatorname{sym} \mathbf{d}\mathbf{u}) = \mathbf{0}$.

Patch-wise weighted residuals problem. Recalling $\boldsymbol{M} := \boldsymbol{p} - \boldsymbol{S}$, we write $\boldsymbol{M}_{\mathbf{u}} = \boldsymbol{p}_{\mathbf{u}} - \boldsymbol{S}_{\mathbf{u}}$ and write the single-field weighted residuals statement as follows. Find $u \in \mathcal{U}^k_{\mathcal{P}}$ such that for every $\mathcal{Q} \in \mathcal{P}_{\mathrm{a}}$

$$\begin{aligned}&\int_{\mathcal{Q}} \hat{\mathbf{v}}_{\mathbf{u}} \wedge (\mathbf{d}\boldsymbol{M}_{\mathbf{u}} - \rho \boldsymbol{b}) \\ &\quad + \int_{\partial\mathcal{Q}} \left[\hat{\mathbf{v}}_{\mathbf{u}} \wedge \left(\overset{\star}{\boldsymbol{M}} - \boldsymbol{M}_{\mathbf{u}}\right) + \left(\overset{\star}{\mathbf{u}} - \mathbf{u}\right) \wedge \hat{\boldsymbol{p}}_{\mathbf{u}} + \left(\overset{\star}{\boldsymbol{\varepsilon}} - \boldsymbol{\varepsilon}_{\mathbf{u}}\right) \wedge \mathbf{i}_t \hat{\boldsymbol{S}}_{\mathbf{u}}\right] = 0 \;\forall\; \hat{\mathbf{u}} \in \mathcal{U}^k_{\mathcal{Q}}\end{aligned} \tag{2.8}$$

in which a superposed ˆ indicates a weighting function and $\mathbf{i}_t$ is the temporal insertion operator. The terms weighted by $\hat{\mathbf{v}}_{\mathbf{u}}$ are the continuous and jump parts of the Equation of Motion in residual form. The terms weighted by $\hat{\boldsymbol{p}}_{\mathbf{u}}$ and $\mathbf{i}_t \hat{\boldsymbol{S}}_{\mathbf{u}}$ are, respectively, the jump-part remnants of eqs. (2.4a) and (2.4b) in residual form.[8]

We emphasize that the extent to which individual components of the jump conditions listed in Systems 2.5 and 2.7 are active depends on the local manifold orientation on Γ^{J}. For example, we have already noted that the displacement jump condition, eqs. (2.5a) and (2.7a), is inactive on material interface trajectories. In general, this orientation dependence is encoded in the structure of the differential forms in each weighted residual term in the boundary integral of eq. (2.8).

Patch-wise weak problem. Application of the Stokes Theorem to (2.8) generates the weak statement of the patch-wise problem. Find $u \in \mathcal{U}^k_{\mathcal{P}}$ such that for every $\mathcal{Q} \in \mathcal{P}_{\mathrm{a}}$

$$\int_{\mathcal{Q}} (\mathbf{d}\hat{\mathbf{v}}_{\mathbf{u}} \wedge \boldsymbol{M}_{\mathbf{u}} + \hat{\mathbf{v}}_{\mathbf{u}} \wedge \rho \boldsymbol{b}) = \int_{\partial\mathcal{Q}} \left[\hat{\mathbf{v}}_{\mathbf{u}} \wedge \overset{\star}{\boldsymbol{M}} + \left(\overset{\star}{\mathbf{u}} - \mathbf{u}\right) \wedge \hat{\boldsymbol{p}}_{\mathbf{u}} + \left(\overset{\star}{\boldsymbol{\varepsilon}} - \boldsymbol{\varepsilon}_{\mathbf{u}}\right) \wedge \mathbf{i}_t \hat{\boldsymbol{S}}_{\mathbf{u}}\right] \;\forall\; \hat{\mathbf{u}} \in \mathcal{U}^k_{\mathcal{Q}} \tag{2.9}$$

We report results from cSDG seismic simulations using this weak formulation in Chapter 4.

2.5 Chapter Summary

The Tent Pitcher algorithm is an advancing front scheme for constructing causal spacetime meshes that reflect the governing system's characteristic structure. In combination with discontinuous Galerkin discretizations, causal meshes localize the cSDG solution procedure to small clusters of elements called patches. The resulting asynchronous, patch-by-patch solution scheme has linear computational complexity in the number of spacetime patches and a natural parallel structure in which adaptive spacetime meshing and solution of system equations interleave at a common granularity. This unique structure enables dynamic parallel–adaptive

[8] A projection operator is applied to $\hat{\boldsymbol{p}}_{\mathbf{u}}$ to prevent undesirable coupling with the $\hat{\mathbf{v}}_{\mathbf{u}}$ terms. See [20].

cSDG solvers, such as the distributed framework described in the next chapter. The cSDG formulation for linear elastodynamics developed in this chapter is the basis of the numerical seismic model introduced in Chapter 4.

Chapter 3

A Distributed Framework for Parallel–Adaptive Implementations of the cSDG Method

3.1 Introduction

In this chapter, we present a distributed parallelization of the cSDG code. Unlike in the shared memory model, computations are performed on processor cores that do not share a common physical memory space and instead, are distributed across the distributed system. This leads to additional complications when compared to the simpler case of shared-memory, previously presented in [33]. The need for message passing and load balancing across distributed nodes must be carefully considered to achieve scalability.

In an effort to reduce computation times, much work has been made to efficiently parallelize finite element (FE) codes. The *domain decomposition method* (DDM) is an extremely popular approach for load balancing and adaptive mesh-generation in many parallel FE codes [34, 35]. Here, the FE problem domain is divided into multiple, smaller subproblems that can be solved simultaneously [36, 37, 38]. In such a method, the decomposition must be done in a manner that minimizes dependence between the subproblems. This way, parallel processors can be assigned to perform computations on a subproblem with minimum communication between processors. Load balancing must also be performed to dynamically redistribute workload evenly among each processor. Popular decomposition schemes include graph-based techniques [39] and geometry-based techniques [40, 41]. One noteworthy method that has considerable research interest is the finite element tearing and interconnecting (FETI) method [41, 42]. This method requires fewer interprocessor communications than traditional domain decomposition schemes by partitioning the spatial domain into a set of totally disconnected subdomains. The global continuity across subdomain interfaces is enforced via Lagrange multipliers rather than explicit interprocessor dependence.

The solution to elliptic and parabolic PDEs generally involves global coupling and the need for domain decomposition techniques outlined above. Hyperbolic PDEs on the other hand have finite propagation speed of disturbances. This allows for more efficient parallelization schemes as coupling between elements is localized. The discontinuous Galerkin (DG) finite element method is particularly efficient for hyperbolic problems wherein discontinuous basis functions are used to formulate the Galerkin approximation [43, 44, 45]. Early works to parallelize the DG method have utilized the DDM [46, 47, 48, 49], thus introducing synchronization barriers. More recent works on parallel DG methods such as those by Xia *et al.* [50], Raj *et al.* [51], and Krais *et al.* [52] continue to utilize the DDM, albeit with greater sophistication and scale. In this dissertation, we have used the unstructured mesh generation capabilities inherent to the "Tent Pitching" method [16, 17] and developed a fully asynchronous parallel method.

An ideal program for parallelization is one that can be divided into sub-problems that involve no communication between the sub-problems. Problems of this type are commonly called *embarrassingly parallel*. A major portion of cSDG calculations satisfies this criterion — specifically, those that involve spacetime mesh generation, the patch-wise finite element solution process, and local adaptive mesh refinement. All of these execute at a common patch-level granularity. Since the solution of a new patch depends only on previously computed data — as guaranteed when respecting the causality constraint (see Section 2.2.1) — there is no need for communication between active patches during the solution process. The formulation requires data communication only for flux exchanges between inflow facets and facets of patch elements and not edges or vertices, and thus has a relatively low communication cost. Finally, the localized patch-by-patch solution technique limits the number of degrees of freedom per solve, ensuring much smaller matrix-vector operations when compared to traditional finite element methods which need to solve large, global matrix equations. For a front mesh of bounded degree and polynomial order, the cSDG method has linear computational complexity. These features make the cSDG method uniquely well-suited for parallel implementation.

Our parallel implementation replaces the standard *Bulk Synchronous Parallel* (BSP) [4] model with an asynchronous *task-based* and *mostly lock-free* solution scheme as described in this chapter. The BSP model involves three main stages: concurrent computation in multiple processes using only data stored locally; global communication across multiple processes to exchange relevant data; and, barrier synchronization where all processes wait until the communication is complete. A common usage of the BSP model is in traditional finite element codes with synchronous time-marching. After an iterative implicit update or an explicit time step has been computed on each parallel process, relevant data needs to be exchanged between subdomains before the next iteration or time-step. Strongly graded mesh refinement in explicit methods and localized nonlinearity in implicit solvers severely impedes efficiency due to severe time-step restrictions dictated by stability or accuracy requirements. The BSP model can become inefficient due to the cost of barrier synchronization which comes in two parts: the cost caused by the variation in the completion times of the computation steps; and the cost of reaching a globally-consistent state in all processes after the communication. The costs associated with these synchronization barriers are substantial in the case when the computation stage in each process varies greatly in completion times.

While a static domain decomposition supports efficient parallel implementation of non-adaptive cSDG models, the restriction to non-adaptive models sacrifices one of the method's most powerful features. Unfortunately, we find even highly dynamic decomposition incapable of maintaining load and data balance in the face of dynamic, fine-grained cSDG adaptive meshing. Re-computing the subdomains requires global communication and a synchronization barrier which suffers from the limitations of the BSP model as described above.

We abandon both the BSP and DDM in our architecture for the distributed parallel–adaptive cSDG implementation and replace them with a asynchronous, task–based framework augmented with a server–client model for managing front-mesh data, our method's only global data structure. Contrary to standard practice, we prioritize dynamic management of data balance and load balance over minimizing communication costs and freely distribute front-mesh data without consideration for spatial coherence. This unorthodox policy enables powerful probabilistic techniques for dynamic load and data balancing that execute at the common patch-level cSDG granularity and keep up with fine-grained cSDG dynamic adaptive meshing. Data clients request and modify fragments of front-mesh data through communication with the front-mesh servers and enable dedicated solver processes to construct patches that can be solved asynchronously and without any further communication. Updates to the global state of the front data are then communicated to the necessary servers after the patch computation.

The organization of this chapter is as follows. For the rest of Section 3.1 we describe the previous efforts to develop the parallel implementation of the cSDG method. Sections 3.2 and 3.3 details the overall parallel architecture, along with the various distributed algorithms implemented to achieve a high level of parallel scalability across the distributed system. In Section 3.4 we describe the experimental set-up and target machine on which we develop and test our method. Finally, we present scaling results and optimizations of the parallel architecture on a large distributed system to demonstrate its feasibility in Section 3.5

3.1.1 Previous Work

Owing to the rich parallel capabilities inherent to the cSDG method, there have been attempts in the past to parallelize the code. This thesis is part of an ongoing, long-term project to develop a robust, scalable parallel implementation of the cSDG method. The first prototype utilized domain decomposition along with the CHARM++ parallel language and run-time system [53] developed by the Parallel Programming Group at the University of Illinois to manage interprocessor communication, load balancing, and the automatic dispatch and scheduling of patches to be solved. While this unpublished work demonstrated promising speed-up for the non-adaptive case, scalability was limited for the adaptive case where the re-computation of the global domain decomposition was too slow to keep the computational load balanced.

A further improvement was performed by optimizing the assembly and solution stages of the cSDG code where sequential optimization, vectorization and OpenMP multi-threading lead to good speedups for shared-memory parallelization [54]. Additionally, offloading computation to a co-processor card based on the Intel Many Integrated Core (MIC) was explored to further accelerate computation. In this model, multiple patches can be formed and buffered in main memory before being transferred to the co-processor where these patches could be assembled and solved in parallel. This execution model could be readily applied to general-purpose computation on GPUs (GPGPU) where the large number of parallel threads could be leveraged for the compute-intensive stage of patch assembly and solution. The work demonstrated limited speedup due to the fact that the mesh generation was not parallelized.

The latest effort to speed-up the method involved parallelization of the meshing and patch generation code using the POSIX Thread (Pthread) parallel execution model in shared memory [33]. Multiple software threads, organized into multiple thread pools, simultaneously and asynchronously build and solve patches. By enforcing the causality constraint on patch duration[1], we guarantee that each thread performs calculations that depend only on previously computed data. Conflicts between threads trying to construct overlapping spacetime patches are resolved through a system of locks on front-mesh vertices. Additionally, improvements to the adaptive meshing scheme allow for local front-mesh refinement and coarsening while maintaining globally conforming triangulation. This work is the first that balances both data and computational load for the parallel–adaptive case, leading to a substantial improvement over past attempts.

The main drawback is that the restriction to shared-memory hosts limits the number of cores and, therefore, the potential speed-up. This chapter addresses this limitation by describ-

[1]We define patch duration as the patches temporal diameter, *i.e.,* the non-negative difference in time coordinate between any two vertices in a patch.

ing a novel architecture for distributed-memory, parallel–adaptive cSDG implementations as well as the various supporting algorithms it requires.

3.2 Distributed Parallel cSDG Architecture

This section describes the distributed parallel cSDG architecture in detail as well as the various algorithms implemented for communication, thread-safety, and load-balancing — some of the main challenges for the distributed code.

3.2.1 Terminology and Notation

Before we proceed to the technical details of our distributed parallel architecture, we define some basic concepts useful to the rest of the chapter. Readers are referred to Section 2.2 for an introduction to the causal spacetime meshing that is key to our parallel architecture.

Distributed Front Mesh

The space-like front (see definition 2.2) mesh consists of *vertices* and *cells* embedded in one or more spacetime manifold. The front at the beginning of each iteration of our algorithm represents the frontier of both the incremental mesh construction as well as the solution. Thus, for any point $P = (p, \tau(p))$ on the front τ, the solution at P has been computed and the set of points $P^+ = (p, t^+)$, such that $t^+ > \tau(p)$, remain to be meshed and the solution there is unknown. The front mesh advances in time as the solution progresses, but at any given stage of the solution procedure, each front vertex is assigned non-uniform temporal coordinates that define the d-dimensional front mesh, analogous to a "bumpy terrain". In this way, the front mesh represents the state of the solution progress and forms the *only* global data structure in the tent pitcher solution scheme.

For the parallel code, the *distributed front mesh* data-structure — consisting of d-dimensional vertices and cells — are uniformly distributed across the distributed system through a yet-to-be defined *partitioning scheme*. At its simplest, a general partitioning scheme attempts to assign equal number of front entities to each subdomain while minimizing communication costs between subdomains. While a rather intuitive and popular choice for this scheme is to partition front entities respecting spatial coherence, in practice maintaining this coherence would require either expensive re-partitioning steps or complicated load and data-balancing schemes to support adaptivity. As described in Section 3.3.6, as long as the cost for communication is minimal and the latency is masked, we can simplify the partitioning by abandoning spatial coherence and implement highly dynamic load and data-balancing schemes that share the same granularity as patch-wise adaptive operations.

Footprint and Extended Footprint

For any vertex v on the front τ, the *footprint* is defined as a collection of front cells and vertices that are incident on v. The vertex v is labeled as the *base* of the footprint. More formally, we define the footprint as follows:

Definition 3.1 (Footprint Δ)**.** For any vertex v, the footprint on this vertex Δ_v is the *star* of v where

$$St(v) = \{\Delta_v \in \tau | \Delta_v \cap v \neq \varnothing\}^{2}. \tag{3.1}$$

The footprint is useful as it is the smallest fragment of the front mesh that is sufficient to support any adaptive operation such as refinement or coarsening.

For the parallel code, it becomes necessary to include more front entities to the definition of the footprint. This is motivated by the fact that many refinement operations can alter connectivity information outside the footprint as defined above. As such, we define the *extended footprint* for a front vertex v as:

Definition 3.2 (Extended Footprint $\tilde{\Delta}$)**.**

$$\tilde{\Delta}_v = \bigcup_{f \in St(v)} \bigcup_{c \in edge(f)} St(c). \tag{3.2}$$

In other words, the extended footprint is the union of the footprint along with all the front entities that are incident to the faces (or edges) in the star of v. Figure 3.1 depicts the extended footprint for a two-dimensional front. We will use the terms footprint and extend footprints interchangeably going forward to solely refer to the extended footprint as this is more applicable when describing the parallel architecture.

Local Time-minimum Vertex

Definition 3.3 (Local minimum vertex)**.** A local time-minimum v_{min} of the front τ is a vertex that satisfies the following:

$$v_{min} = \{p \in \tau \ \mid \tau(p) \leq \tau(q) \ \ \forall q \in Lk(p)\}, \tag{3.3}$$

[2] In some definitions, the star is not closed under taking faces. To remove ambiguity, we more precisely define the footprint as the *closed* star which is a closed simplicial complex formed by adding all the missing faces to $St(v)$

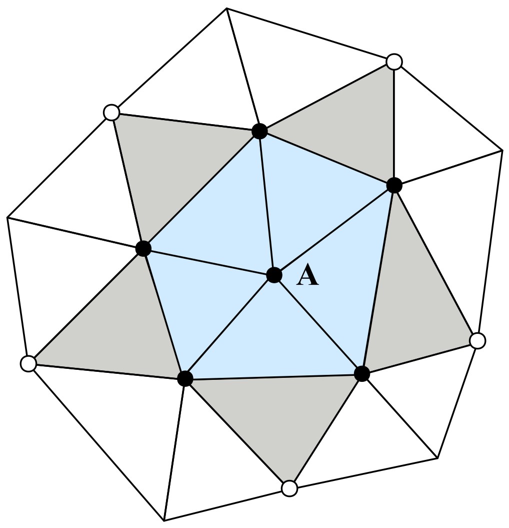


Figure 3.1: Footprint (blue regions) for a 2 dimensional front over vertex A with link vertices ($\bullet$), along with the extended footprint (union of grey and blue regions) and extended vertices ($\circ$)

where the link operator $Lk(\cdot)$ is defined as all vertices in the open star of v that is disjoint from v, *i.e.,*

$$Lk(v) = \{v' \in St(v) | v' \cap v = \varnothing\}. \tag{3.4}$$

Local time-minimum vertices are important in the tent pitcher algorithm as patches that are pitched over them are guaranteed to make progress in advancing the front [30]. Every front has a local minimum because it has at least one global minimum which is also a local minimum.

Pitchable Front Vertex

For a given parallel process, a pitchable front vertex v is a vertex on the current front that satisfies the following criteria:

1. it is a local time-minimum vertex, *cf.* Def. 3.3
2. all vertices in its extended footprint, $\tilde{\Delta}_v$, are *owned* by that process

A vertex is "owned" by a parallel process if it has exclusive rights to that vertex, very similar to that of a lock or mutex. We will formalize this concept later in this chapter through the lock and hold algorithm (see Section 3.3.1).

3.2.2 Overall Architecture

The distributed parallel architecture is implemented to leverage the embarrassingly-parallel structure of the overall cSDG algorithm. An overall *task-based* framework, augmented with a *server-client* execution model, is chosen for the parallel architecture wherein the clients request data from distributed servers in order to make progress in the simulation by generating tasks for asynchronous FE solvers. Communication between the clients and servers across the distributed system is handled by the Message Passing Interface (MPI) framework.

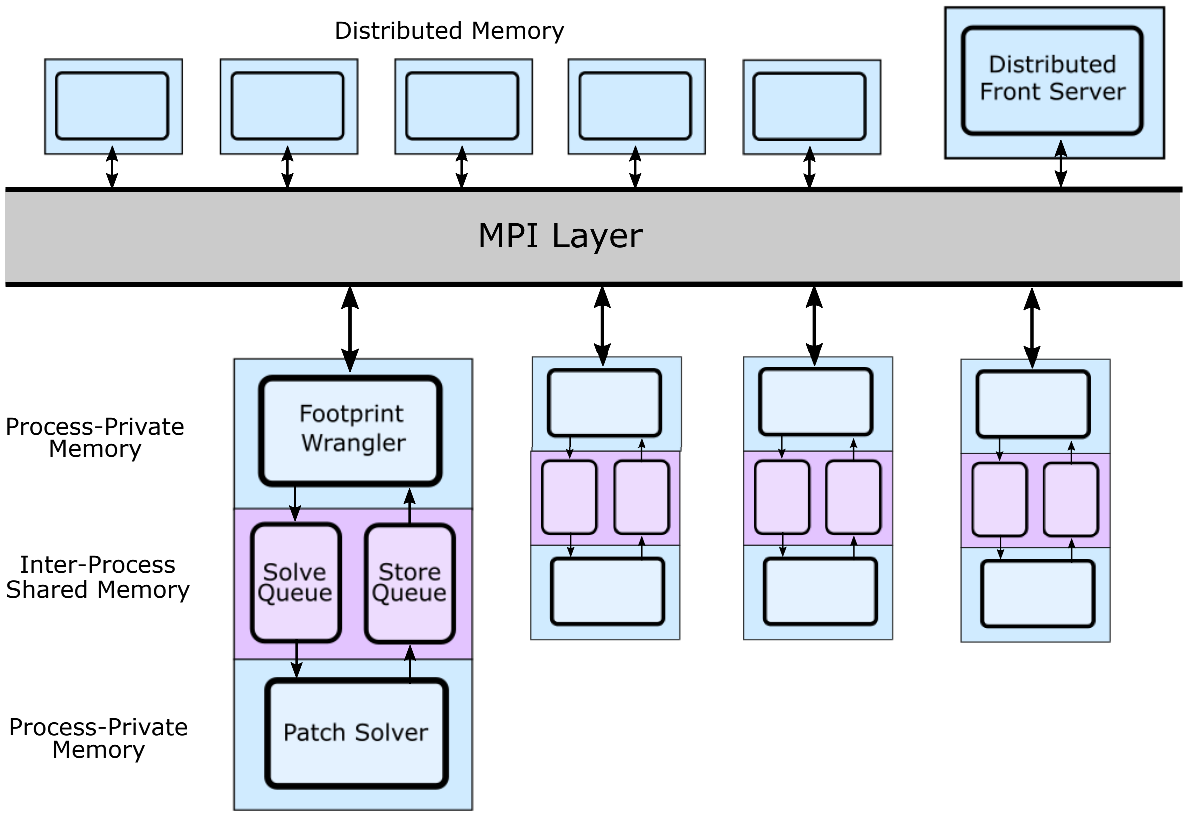


Figure 3.2: Distributed parallel cSDG architecture depicting the three parallel process types and client-server communication pattern using the MPI framework.

Figure 3.2 shows the overall software architecture of the distributed parallel implementation which entails three class of asynchronous processes. The distributed front mesh — our only global data structure — is partitioned across distributed memory and managed by server processes called *distributed front servers*. The client processes, the *footprint wranglers*, assemble extended footprints by requesting the necessary front entities from the relevant front server. The footprints contain all necessary data for the embarrassingly parallel solution scheme: adaptive front refinement, adaptive construction and solution of spacetime patches, computation, and checking of adaptive error indicators. Once the footprint is assembled in local memory by the footprint wrangler, it is then pushed onto a *solve queue* that feeds work to a paired *patch solver* processes. Each solver process pitches a patch over the footprints in this queue and can complete assembly, solve and adaptivity checks without any further communication or synchronization with other processes using data exclusively in process-private memory. Therefore, we expect perfect scaling if (i) there is always work in the solve queue and (ii) if the solvers are isolated from indirect interference by the server and wrangler processes. The first requirement is easily satisfied by common latency-hiding techniques, but solver isolation is harder to achieve. We investigate these issues later in this chapter. After the solution has been computed, the solver returns the updated footprint and solution data via a *store queue* to its paired wrangler which outputs solution data and communicates with the servers to update the distributed front. The number and distribution of each of these processes vary from system-to-system, but as a general rule, we aim to maximize the system resources allocated to the patch solvers while keeping the latency between message request and response manageable.

We present algorithmic details of each of the three distinct parallel process types — (i) distributed front servers, (ii) footprint wranglers, and (iii) patch solvers — in the following sections.

3.2.3 Distributed Front Servers

The front servers store, update, and respond to requests for the distributed front mesh data structure (defined in Section 3.2.1). We make use of *non-blocking* MPI communication semantics which ensures that the server processes regains control after initiation of message transmission. By careful design of the server algorithm, waiting for data to arrive and the actual transfer of data occurs in the background, leaving resources free to process other requests.

Table 3.1: Data members for distributed server process

Name	**Description**
Front vertices	Partitioned front vertices assigned to the server processes. Part of the local front mesh fragment
Front chambers	Partitioned front chambers/cells assigned to the server processes. Part of the local front mesh fragment
end_cond	Boolean attribute updated by a central process to indicate simulation has completed

The server process maintains the local fragments of the distributed front mesh as detailed in Table 3.1. The two main types of operations performed on the servers can be classified into those that *impact the state* of the local data and those that do not. Examples of the first type are: (i) write operations that modify currently stored data, (ii) insert operations that add a new entity to the local front mesh, and (iii) delete operations that remove an existing entity from the local front mesh. These operations are typically performed during the store stage when wranglers send the updated footprint information to the servers. Operations that do not impact the global state are read requests where wranglers request front entities during footprint construction.

The overall structure of the distributed front server processes is presented in Algorithm 1. Server processes each initialize their local, partitioned, fragment of the front mesh as described later in Section 3.3.6 during the initialization stage. This involves reading and extracting the portions of the initial front mesh that are mapped to the local server process. The servers then loop until signaled to shut-down and continuously respond to any request made via MPI non-blocking requests. Based on the type of request received, we either update the local fragment of front data and termination variables (further discussed in Section 3.3.8) or send the requested front data to the requesting wrangler.

Algorithm 1 Server Pseudo-code

```
1:  Initialize local distributed front mesh fragment
2:  end_cond ← false
3:  while !end_cond do
4:      Perform MPI_IProbe() to check for incoming messages
5:      if message exists then
6:          Perform MPI_IRecv() and process message
7:          if message type = Read then
8:              Access local fragment of distributed front mesh
9:              Perform MPI_ISend() to requesting wrangler rank with data
10:         else if message type = Write/Insert/Remove then
11:             Update local fragment of distributed front mesh
12:             Update termination variables, see Algorithm 15
13:         end if
14:     end if
15:     Update end_cond
16: end while
17: Finalize and shut-down
```

3.2.4 Footprint Wrangler Clients

The footprint wrangler clients are responsible for gathering and constructing extended footprints that are the basis of all parallel computation in the cSDG method. The wranglers build footprints over local time-minimum vertices in the front. In the parallel code, each wrangler maintains its own list of unique time-minimum vertices that are initialized as a subset of the initial front mesh and then updated as the simulation progresses. As in the case of the server processes, we utilize non-blocking MPI communication semantics for communication with the servers to maximize the time spent doing useful work.

Each wrangler process maintains the data members outlined in Table 3.2. Local minimum vertices over which footprints are constructed are taken from the *minimum vertex heap* (MVH), a time-ordered priority min–heap. We maintain a further heap — the *pending vertex heap* (PVH) — the need for which becomes apparent by considering thread-safety when multiple wrangler processes are simultaneously attempting to construct footprints over the front. We utilize a system of *vertex locks* on each front vertex to prevent collisions between multiple wrangler processes, as detailed in Section 3.3.1. For now, it is sufficient to view the construction of a footprint over a local minima as an operation that could fail due to a failure to acquire all the vertex locks across the entire extended footprint. When such failures occur, the local minima vertex that formed footprint base is added to the PVH instead of the MVH to prevent unproductive re-checking of footprints before all their vertices become unlocked. After some non-negligible time has elapsed, vertices added to the PVH can be moved to the MVH for reconsideration as the state of the locks on the front can be expected to change. The management of these heaps are described in greater detail in Section 3.3.5.

Table 3.2: Data members for distributed wrangler process

Name	**Description**
Minimum vertex heap	Time ordered priority heap of local minima on the front mesh allocated to the wrangler process.
Pending vertex heap	Time ordered priority heap of local minima on the front mesh over which footprint construction has failed.
Solve queue	FIFO queue for communication between wrangler and solvers containing fully assembled footprints over which solvers can pitch and solve patches
Store queue	FIFO queue for communication between wrangler and solvers containing footprints updated after solution on the solvers
end_cond	Boolean attribute updated by a central process to indicate simulation has completed

Each wrangler client process has a *paired* patch solver process which communicates using the *solve* and *store queues*. When a footprint has been successfully assembled by the wrangler (by gathering all front entities and successfully acquiring all vertex locks in the footprint), it is pushed onto the solve queue for patch construction and solution by the solver process. The base vertex of the footprint is necessarily a pitchable front vertex, as defined in Section 3.2.1. Once the footprint has been assembled, there is no further need for global front data and the downstream tasks performed by the solver are *embarrassingly parallel*. After the solver has completed the patch solution, the updated footprint is placed onto the store queue to be scattered by the wrangler to the relevant front server. Additionally, after the pitch operation the updated time-coordinate of the footprint base vertex may cause neighbor vertices to become new local time-minima. We utilize the scheme outlined in Section 3.3.4, which operates at the same granularity of the patch, to check for such new local minima to replenish the minima vertex heaps. The pseudo-code for the wrangle and store operations are presented in Algorithms 2 and 3 respectively. The wrangler process can be implemented with the wrangle and store operations occurring in sequence or simultaneously using multiple threads. We find no difference in performance for either case. Both these operations are implemented internally as a finite-state machine (FSM) where each stage of footprint wrangling/storing are discrete states of this state machine. Due to the latency between requesting front entities from the server process and receiving the response, the wrangler process continuously loops and tries to progress the internal state of the wrangle and store operations based on the messages received and processed. When we transition to a new state, new messages are sent to the relevant servers to request or send the front entities needed for that state. The full description of the stages of the wrangle and store FSMs are given in Section 3.3.3. Similar to the server processes, the wrangler continuously performs work until signaled to terminate at the end of the simulation using a globally-updated condition variable.

Because the time-minimum vertices are analogous to a list of available tasks the wrangler can perform, we must be very careful to distribute these vertices evenly across the distributed system. If a subset of wranglers exhaust their local list of time-minimum vertices, these wranglers will be unable to build footprints and, in turn, their associated solver process will remain idle. This condition is called a *solve queue stall* since the solve queue shared between the wrangler and solver empties due a lack of work on the upstream side. We closely monitor the state of the solve queues and tune the parameters of the parallel code to minimize the occurrences of solve queue stalls. Additionally, we implement a highly dynamic, probabilistic load-balancing scheme to balance the distribution of time-minimum vertices on each wrangler, as detailed in Section 3.3.7.

3.2.5 Embarrassingly Parallel Patch Solver

The patch solver process builds patches over footprints stored in its associated solve queue and performs the finite element assembly, solution, and adaptive operations on the patch.

Algorithm 2 Wrangle Pseudo-code

```
1: Initialize local minimum_vertex_heap, pending_vertex_heap, and solve_queue
2: is_wrangling ← false
3: end_cond ← false
4: while !end_cond do
5:     if !is_wrangling AND solve_queue not full then
6:         Pop vertex from minimum_vertex_heap, see Algorithm 10
7:         is_wrangling ← true
8:     else if is_wrangling then
9:         Perform MPI_IProbe() to check for incoming messages
10:        if message exists then
11:            Perform MPI_IRecv() and process message
12:            Load data into footprint and progress state, if possible
13:        end if
14:        if footprint state transitioned to new state then
15:            Identify distributed data required to progress footprint state
16:            Perform MPI_ISend() to relevant server rank requesting data
17:        end if
18:        if footprint wrangling complete then
19:            is_wrangling ← false
20:            if locking successful then
21:                Add footprint to solve_queue
22:            else
23:                Reset all locks acquired during wrangling
24:                Return base vertex to pending_vertex_heap, see Algorithm 10
25:            end if
26:        end if
27:    end if
28:    Update end_cond
29: end while
30: Finalize and shut-down
```

Algorithm 3 Store Pseudo-code

```
 1: Initialize local minimum_vertex_heap, and store_queue
 2: end_cond ← false
 3: while !end_cond do
 4:     if store_queue has data then
 5:         Pop footprint from store_queue
 6:         Perform MPI_ISend() to relevant server rank with updated footprint data
 7:         If any vertices are local minima, push onto minimum_vertex_heap, see Algorithm 9
 8:     end if
 9:     Update end_cond
10: end while
11: Finalize and shut-down
```

The solver process follows the *task-based execution model* where a process (or a group of parallel processes) continuously poll a task queue for work and executes the task as soon as one is received. Once complete, the process return to polling the task queue. The data members for the solver process are listed in Table 3.3. As described in Section 3.2.4, the solve and store queues are used to transmit footprints from and to the paired wrangler process, respectively. Fully assembled footprints are popped from the solve queue as soon as they become available and the patch can be built and solved as outlined in Algorithm 4. By respecting the causality of the underlying method, all the data needed to perform these operations are now present in the solver's process-private memory. There is no need for further communication or synchronization at this stage. As long as there are tasks present in the solve queue and space in the associated store queue, the patch solvers can proceed uninterrupted. Through the separation of MPI communication and patch solution between the wrangler and solver processes, we hide the latency of MPI communication across the distributed network from useful computation.

Table 3.3: Data members for solver process

Name	**Description**
Solve queue	FIFO queue for communication between wrangler and solvers containing fully assembled footprints over which solvers can pitch and solve patches
Store queue	FIFO queue for communication between wrangler and solvers containing footprints updated after solution on the solvers
end_cond	Boolean attribute updated by a central process to indicate simulation has completed

Algorithm 4 Solver Pseudo-code

```
1: Initialize local solve_queue, store_queue
2: end_cond ← false
3: while !end_cond do
4:     if solve_queue has data then
5:         Pop footprint from solve_queue
6:         Build patch over footprint
7:         Solve patch and perform necessary adaptive operations
8:         Store updated footprint onto store_queue
9:         Update end_cond
10:    end if
11:    Update end_cond
12: end while
13: Finalize and shut-down
```

An important goal for the distributed parallel code is to minimize the cases where solver processes halt execution due to starvation of tasks from the solve queue, a condition known as a solve queue stall. Multiple factors contribute to solve queue stalls:

1. A small number of pitchable front vertices relative to the global front size. This is common during the initial and final stages of the simulation where the number of total vertices on the active front is limited. Due to conflicts between multiple competing wranglers, only a small number of local minima vertices satisfy the conditions to be pitchable.

2. Large latency in communication between servers and wranglers due to network congestion or over–saturation.

3. Insufficient resources allocated to the server and wrangler processes. Since these processes are responsible for generating tasks for the solvers, a poor allocation of resources could lead to insufficient footprint generation.

Many design decisions described later in this chapter are based on the objective to minimize the number of solve queue stalls.

3.3 Distributed Algorithms

In this section we provide further details on the various algorithms implemented for the distributed-parallel code outlined above.

3.3.1 Lock and Hold Management

In the parallel setting, we may encounter collisions from another wrangler process while attempting to build footprints. A collision occurs when the extended footprint built by one process extends into the extended footprint of another process. Failure to establish well-defined algorithms to resolve the collisions would lead to race conditions and thread-safety issues. Particularly in the case of adaptivity, data corruption can occur if multiple processes attempt to modify the same portion of the front at the same time. We utilize vertex *locks* to establish clear ownership of front mesh vertices to one process over another. In addition, we favor algorithms that deliver sequences of patch solves that are reasonably close to strict ascending-time order. We utilize vertex *holds* to prioritize pitches over vertices that have smaller time coordinate. We introduce two members for every front vertex object, a lock and a hold object, as defined below:

Definition 3.4 (Vertex Lock)**.** A lock object is a logical-valued member of every front vertex object that indicates the ownership of the vertex. An active lock signifies that a front vertex is unavailable for use in another footprint.

Definition 3.5 (Vertex Hold)**.** A hold object is a member of every front vertex object that tracks priority rights when collisions occur between footprints. The hold has the following members:

- Base vertex ID: the ID of the footprint base vertex that set the current hold and,
- Base time: the time-coordinate of the footprint base vertex that set the current hold,

to maintain a close to ascending-time front mesh for parallel efficiency.

A vertex is locked if the owning process has exclusive rights over the vertex and can use the vertex in constructing footprints. A vertex is held if the owning process has priority rights when collisions occur (as detailed in Definition 3.6). Finally, a vertex is *blocked* if it is already locked or another process has a higher priority hold over it. A hold associated with the vertex's native process does not trigger a blocked condition.

When traversing vertices belonging to a footprint under construction on wrangler processes, each vertex must first pass a *block test*, the result of which will determine if the vertex is blocked. This is essential to determine if a footprint constructed on another wrangler process has priority over the native footprint and should be favored to proceed at the expense of the native process. The block test is outlined in Figure 3.3 while the algorithm for the same is presented in Algorithm 5. For a given vertex v in the current footprint, the first check in this test is the state of the lock. A locked vertex necessarily means that a vertex is blocked — a

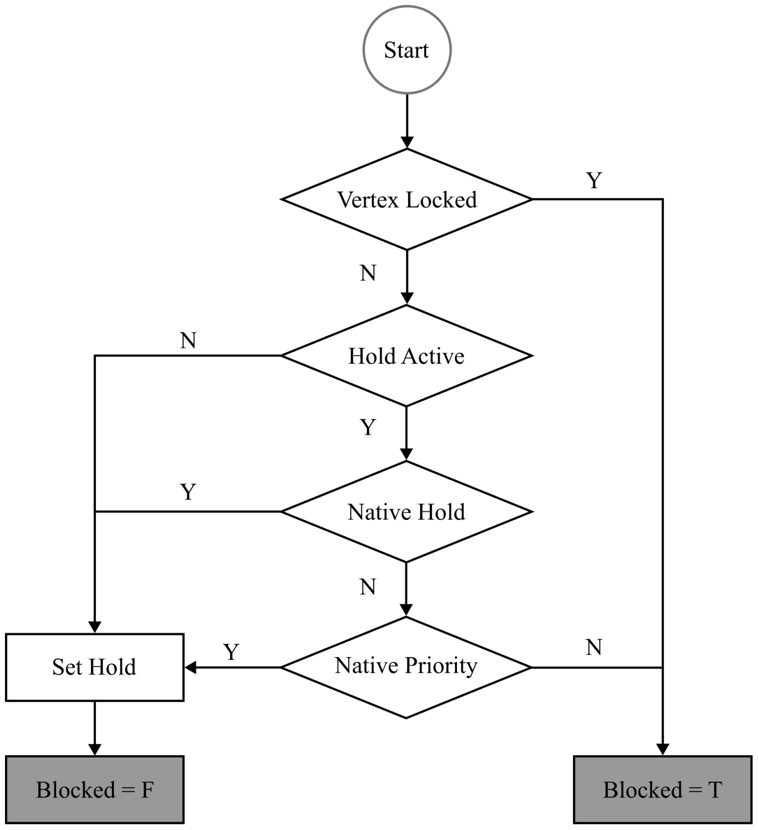


Figure 3.3: Flowchart depicting block test for a vertex.

foreign footprint has ownership of the vertex. If v has no active hold, or if the hold is native to the footprint base vertex, the native process sets the hold and the vertex is not blocked. However, if an active hold already exists for v — as in the case for a collision between two wrangler processes — the priority of the native hold must be compared to that of active hold object. If the native hold has priority, then the current hold is over-written with the native hold and the vertex is not blocked. Otherwise, the vertex is blocked.

If the vertex is not blocked, the native process can acquire its lock. This process is repeated for all vertices in the footprint or until we encounter a blocked vertex. Only once a process can successfully acquire all the locks in the footprint can we pitch over the base vertex. The base vertex satisfies the criteria for a *pitchable vertex* at this stage. Since each vertex is exclusively partitioned to a single server process this naturally enforces the atomicity properties that are essential to the logic of this algorithm. In particular, a server process can only process one request to check the lock status on front data that it controls at a time.

Causality constraints determine a partial ordering of front-mesh vertices that governs the sequence of patch construction and solution in the cSDG solution scheme. Further, we only construct patches over vertices that are local time-minima. Among the many sequences

Algorithm 5 Pseudo-code for block-test on vertex

INPUT: `v_id` - Vertex ID to check; `base_id` - Vertex ID of footprint base; `base_time` - Time coordinate of footprint base

OUTPUT: `vert_blocked` - Result of block test

```
1:  procedure BLOCK TEST(v_id, base_id, base_time)
2:      vert_blocked ← false
3:      hold ← get_vert_hold(v_id)
4:      if v_id locked then                    ▷ If the vertex is locked, then we must be blocked
5:          vert_blocked ← true
6:      else if hold inactive then             ▷ If the current hold is inactive, update it
7:          hold ← active
8:          hold.state() ← (base_id, base_time)
9:      else
10:         if hold.owner() != base_id then     ▷ Previous hold set by foreign footprint
11:             new_hold.state() ← (base_id, base_time)
12:             if new_hold < hold then         ▷ Current hold has priority, see Definition 3.6
13:                 hold ← new_hold
14:             else                            ▷ Previous hold has priority
15:                 vert_blocked ← true
16:             end if
17:         end if
18:     end if
19: end procedure
```

that satisfy these requirements, we prefer those that construct patches over local-minimum vertices in ascending-time order. This preference is easily attained in serial calculations, but we relax it to enable efficient asynchronous computations in the parallel case. That is, we seek parallel algorithms that deliver patch sequences that are close to, but not strictly in, ascending-time order. Large deviations from the ideal ordering can introduce *implicit locks* that impair parallel performance and can make it harder to satisfy causality constraints in nonlinear problems. For instance, if large portions of the front were pitched far ahead of other portions, the number of local minima and, in turn, pitchable front vertices on the front would be much smaller than for a front that is close to ascending-time order. This would lead to undesirable solve queue stalls as discussed in Section 3.2.5.

To maintain fronts that are close to ascending-time order, we define the priority of the hold so as to consider the time coordinates of the footprint base vertex that set the hold. In cases of collisions, the following test is performed to determine which hold has priority:

Definition 3.6 (Hold priority)**.** Given two hold objects h_1 and h_2 with active states (id_1, t_1)

and (id_2, t_2) respectively, we say that h_1 has a priority over h_2 if either of the following hold:

$$t_1 < t_2 \tag{3.5a}$$

$$t_1 = t_2;\ id_1 < id_2 \tag{3.5b}$$

In this way, if collisions do occur, the footprint with the earlier in time base vertex will have priority and the resulting front will satisfy the condition of pitching vertices in ascending-time order. We use the base vertex's id in the case of a tie.

When a patch has been successfully solved, part of the update process performed by the wrangler is to release all the vertex lock and hold objects in the updated footprint. While resetting the locks involves simply setting the logical value to inactive, when resetting the holds we retain higher-priority holds in footprints for reuse later on as described in Algorithm 6. In this way any higher-priority holds set after the vertex has been locked by the native footprint are retained.

Algorithm 6 Pseudo-code for resetting vertex hold

INPUT: `v_id` - Vertex ID to reset hold; `base_id` - Vertex ID of footprint base; `base_time` - Time coordinate of footprint base

```
1: procedure RESET HOLD(v_id, base_id, base_time)
2:     hold ← get_vert_hold(v_id)
3:     if hold.owner() = base_id then        ▷ Only reset if hold is set within the current footprint
4:         hold.state() ← ()
5:         hold ← inactive
6:     end if
7: end procedure
```

3.3.2 Abstract Parallel Driver

Each of the parallel processes described in Sections 3.2.3 - 3.2.5 operates with the same high-level pattern — continuously poll the internal state of a finite-state machine and attempt to make progress to advance the state. To simplify the development of these processes, we define an *abstract parallel driver* class that is used as an abstract base class from which we derive each of the parallel processes.

The parallel driver class, depicted in Algorithm 7, is a wrapper over a parallel *thread* that can asynchronously loop over high-level abstract *tasks* and check and progress the internal state of each task through the *main loop* method. When all the tasks are completed, the driver will finalize and gracefully exit.

Algorithm 7 Parallel Driver Pseudo-code

```
1: procedure INITIALIZE DRIVER( )
2:     Add each high-level task to tasks
3:     for each task t in tasks do
4:         t.initialize()
5:     end for
6:     Initialize and launch thread
7: end procedure

Ensure: size of tasks, sub_tasks $\neq 0$
8: procedure MAIN LOOP( )
9:     while all tasks not done do
10:        for each task t in tasks do
11:            t.check_for_completeness()
12:        end for
13:    end while
14:    for each task t in tasks do
15:        t.finalize()
16:    end for
17:    Notify threads to shut-down
18: end procedure
```

Each task is represented internally as a FSM with the state variables: state, s and the number of sub-tasks completed at that stage. Each abstract *sub-task* is represented by two purely virtual functions that need to be defined in the derived class:

1. `initialize()`: Initializes the members and internal state of the sub-task,
2. `is_subtask_complete()`: Check completeness and try to progress internal state of sub-task. Returns a Boolean value indicating whether the sub-task has been completed.

Once initialized, the task sequentially queries each sub-task at the current state and checks for completeness of the sub-task. Once all sub-tasks are completed at the current state, the task transitions to the next state, as depicted in Algorithm 8. These high-level abstractions enable us to represent asynchronous communication patterns between processes cleanly and with low-level implementation details out of sight.

3.3.3 Wrangler finite-state machines

The wrangler process derives two high-level tasks defined in Section 3.3.2, the *wrangle* and *store* tasks to perform footprint construction and storage respectively. Each of theses tasks

Algorithm 8 High-level task Pseudo-code

```
procedure INITIALIZE TASK( )
    Add each sub-task: (state s, high-level subtask st) to sub_tasks
    for each sub-task state s in sub_tasks do
        for each sub-task st in state s do
            st.initialize()
        end for
    end for
end procedure

procedure CHECK FOR COMPLETENESS( )
    if state = Complete then                    ▷ no more work to do for current task
        return
    end if
    for each sub-task st in current state state do
        st_complete ← st.is_complete
        if !st_complete then
            st_complete ← st.is_subtask_complete()
            num_subtasks_complete ← num_subtasks_complete + st_complete
        end if
    end for
    all_subtasks_complete ← (num_subtasks_complete == st.size())
    if all_subtasks_complete then                    ▷ transition to next state
        all_subtasks_complete ← false
        num_subtasks_complete ← 0
        state ← next state
    end if
end procedure
```

are represented as a FSM where each state is represented as a sub-task defined below. The full pseudo-code for these state machines are present in Appendix A for brevity.

Wrangle Task

The wrangle task is responsible for constructing footprints over pitchable front vertices. Once a local time-minimum vertex has been selected from the minima vertex heap, we activate the wrangle task and continuously check and progress the state. Since none of the required front data is present in process-private memory of the wrangler, at each stage of the FSM requests are made to the relevant server processes and all responses need to be received before transitioning to the next state. The overall wrangle task is depicted in Figure 3.4. The footprint is constructed by first requesting the base vertex data. Once received, the entire extended footprint is gathered by requesting — in order — the incident chambers, link

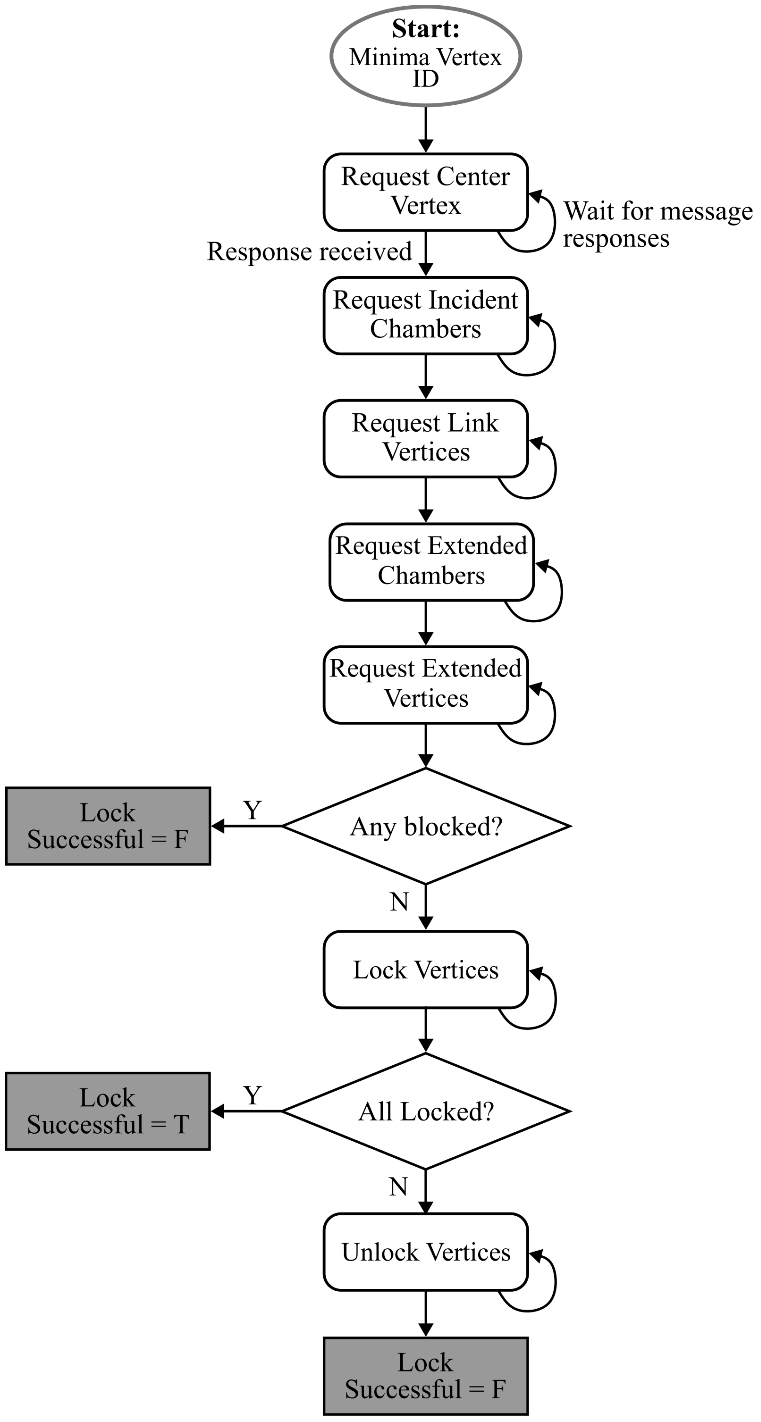


Figure 3.4: Flowchart depicting wrangle task. For each state, data is requested from server processes and all responses need to be received before transitioning to the new state. Based on the state of the holds and locks on the footprint vertices, the footprint construction can either succeed or fail.

vertices, extended chambers and extended vertices in a similar fashion. For each vertex thus processed, we test its block status *cf.* Algorithm 5 and return the result to the requesting wrangler. If all vertices in the footprint are not blocked, we proceed with the footprint construction by requesting all the vertices to be locked. We check the block status of all the vertices in the extended footprint even if we encounter a previously blocked vertex to ensure we correctly set the priority on all footprint vertices. Since some non-negligible time has passed since initially checking the locks when performing the block test, we must ensure that all vertices can be successfully locked. If any vertex is locked before the native footprint can acquire them, we must unlock all the vertices we *have* successfully acquired to ensure a failed footprint resets the lock state.

Store Task

The store task performs the store of the footprint after a solve by scattering the updated front entities — along with the computed solution stored on these entities — to the relevant server processes, as depicted in Figure 3.5. The store involves up to four operations for front entities, if applicable, performed in the following order: erase, insert, update and unlock. Besides the insert operation, none of the store operations require a response from the server process and the store state can be transitioned as soon as the requests are made. Insert operations are different since the server onto which to perform the insert is a non-trivial choice. In order to balance the number of requests made to each server process, we strive to maintain close to equal number of front entities on each server. The choice of server is made in accordance to a data-balancing scheme — detailed in Section 3.3.7 — to achieve this goal. Once the server is selected, the ID of the inserted entity can only be generated by that server based on the generation algorithm (see Algorithm 12 for details). The *global ID* thus generated by the remote server is then used to update the local ID of the entity to be inserted as well as the connectivity of front entities in the footprint. Finally, once all local IDs have been replaced with global ones, we send all impacted front entities to the respective server to update. As part of the update operation, each vertex that is a new local minima, identified as outlined in Section 3.3.4, is added to the minima vertex heap of the favorable wrangler process determined using the load-balancing algorithms presented in Section 3.3.7. The final stage of the store task is to reset the locks and holds on the footprints to allow further progress to be made over this updated footprint.

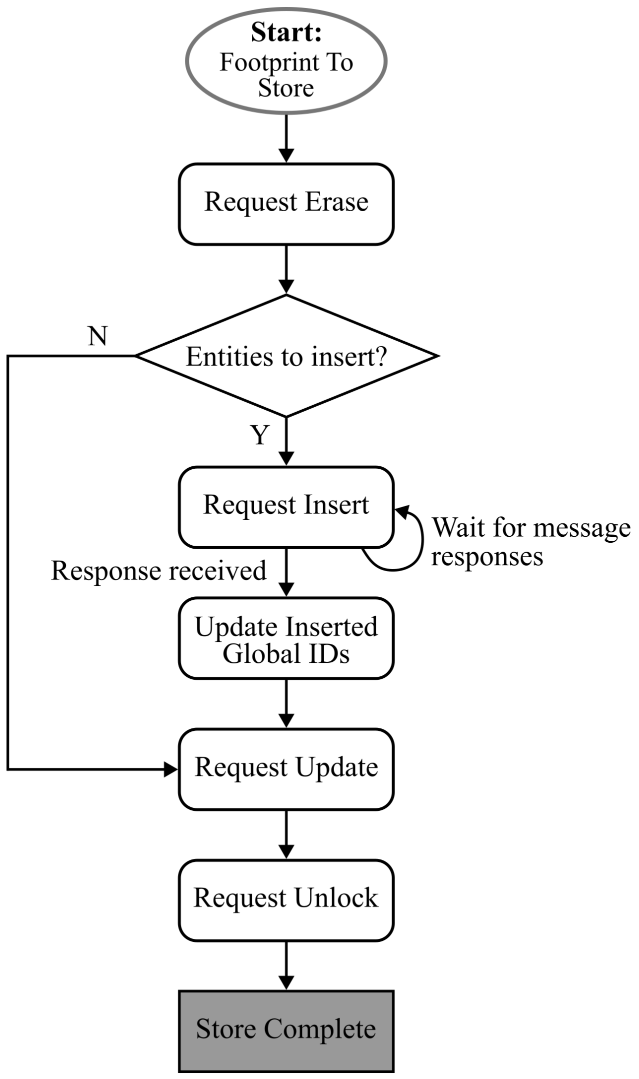


Figure 3.5: Flowchart depicting store task. For each state, the updated front entities of the solved footprint are scattered to the relevant server process to update the distributed front mesh data structure.

3.3.4 Choosing Local Minima Vertices in Parallel

An integral part of the cSDG solution algorithm is determining and updating local time-minimum vertices on the front since we only construct patches over such vertices to ensure progress can be made to subsequent patches. A naive, but valid, strategy of updating local minima is to maintain a time-ordered min-heap of all local minima on the front that is periodically replenished by scanning all front vertices. Aside from the need for a synchronization step that would be prohibitively costly in parallel, this approach would not work in the distributed case where the front data is partitioned among server processes.

For the distributed-parallel code we introduce the concept of *earlier vertex counter* defined as follows:

Definition 3.7 (Earlier vertex counter)**.** Given the local front τ in the neighborhood of a front vertex p, the earlier vertex counter $\mathcal{E}$, is an integer-valued member of p defined as

$$\mathcal{E}_p = \sum_{q \in Lk(p)} [\tau(p) > \tau(q)]\,, \tag{3.6}$$

where $[\cdot]$ is the Iverson bracket operator. In other words, $\mathcal{E}$ indicates the number of link vertices (*cf.* Equation (3.4)) that are strictly earlier in time than vertex p.

In this way, each front vertex has a count of link vertices that are earlier in time and an equivalent condition to Definition 3.3 for a local time-minimum vertex becomes:

Definition 3.8 (Local minimum vertex)**.** A local time-minimum vertex, v_{min}, of the front τ is a vertex that satisfies the following:

$$v_{min} = \{p \in \tau \ \mid \mathcal{E}_p = 0\}. \tag{3.7}$$

Since we construct the extended footprint in process-private memory before each patch solve, the link vertices to the footprint base vertex can be accessed to update $\mathcal{E}$ using the above definitions.

Let τ_i denote the front after the i^{th} step of the Tent Pitcher algorithm. For the initial front τ_0, all vertices are local time-minima since they satisfy Definition 3.8. After the first patch has been pitched and the front has been updated to τ_1, the earlier vertex counter of the base vertex needs to be incremented by the number of link vertices. This vertex will become a new local minima only when all the link vertices are pitched equal to, or past its updated time coordinate.

For an arbitrary front τ_i, a pitch over a front vertex may cause the earlier vertex counters for any subset of the link vertices to change, as depicted for the case of a 1D $\times$ time front in Figure 3.6. In fact, we end up with three distinct cases to consider, independent of dimension. The first is when the updated time coordinates of the base vertex is still earlier than any link vertex (Fig. 3.6(a)). None of the earlier vertex counters are updated for this case. The next case is when the base vertex is updated to be co-equal to a subset of link vertices (Fig. 3.6(b)). Here, only the link vertex that is co-equal to the updated base vertex has its earlier vertex counter decremented by one. Finally, if the base vertex is pitched past a subset of link vertices (Fig. 3.6(c)), the earlier neighbor counter for the base is incremented by one and decremented by one for each such link vertex. This procedure is generalized in Algorithm 9.

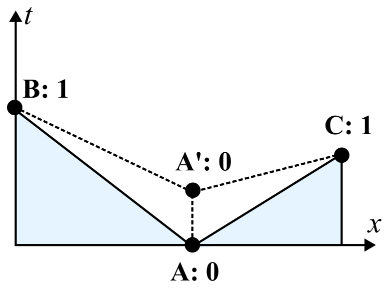
Algorithm 9 Pseudo-code to update earlier vertex counter after a pitch operation

INPUT: `v` - Front vertex corresponding to tent-pole top; `t_base` - Time coordinate of tent-pole base vertex; `t` - Time coordinate of tent-pole top vertex; `ev_ctr` - Earlier vertex counter of vertex

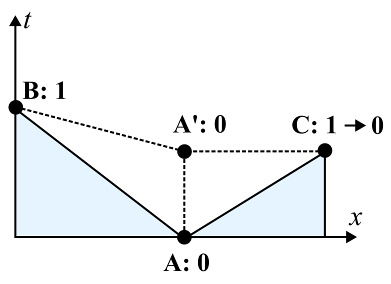
OUTPUT: `ev_ctr` - Earlier vertex counter of vertex

Ensure: `ev_ctr` $= 0$

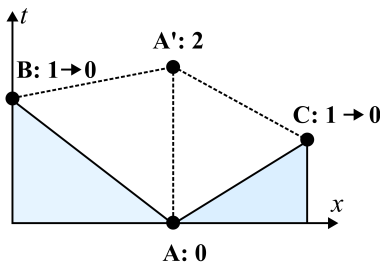
```
 1: procedure UPDATE EARLIER VERTEX COUNTER(v, t_base, t, ev_ctr)
 2:     for unique front vertices nbr_v incindent on v do
 3:         t_nbr ← nbr_v.get_time_coordinate()
 4:         ev_ctr_nbr ← nbr_v.get_earlier_vertex_counter()
 5:         if t ≤ t_nbr then
 6:             if t = t_nbr then                          ▷ v and nbr_v are co-equal
 7:                 ev_ctr_nbr ← ev_ctr_nbr - 1
 8:                 if ev_ctr_nbr = 0 then
 9:                     Mark nbr_v as a new local minima
10:                 end if
11:             end if
12:         else                                             ▷ v is pitched past nbr_v
13:             ev_ctr ← ev_ctr + 1
14:             if t_base < t_nbr then        ▷ Original position of v was earlier than nbr_v
15:                 ev_ctr_nbr ← ev_ctr_nbr - 1
16:                 if ev_ctr_nbr = 0 then
17:                     Mark nbr_v as a new local minima
18:                 end if
19:             end if
20:         end if
21:     end for
22:     if ev_ctr = 0 then                ▷ v is still a local minimum after the pitch op
23:         Mark v as a new local minima
24:     end if
25: end procedure
```



(a) Tent-pole top time-coordinate is less than any neighbor. No change in any earlier vertex counters, see Algorithm 9, Line 5



(b) Tent-pole top time-coordinate is equal to neighbor. Decrement this neighbor's earlier vertex counter, see Algorithm 9, Line 6



(c) Tent-pole top time-coordinate is greater than neighbor. Decrement neighbor's earlier vertex counter and increment tent-pole's counter, see Algorithm 9, Line 14

Figure 3.6: Two successive 1D × time fronts, ABC and $A'BC$, formed after a pitch over tent-pole base vertex A, depicting updates for earlier vertex counter. Earlier vertex counters for each vertex before and after tent-pitch are presented for each vertex with $\rightarrow$.

Besides a pitch operation, adaptive operations such as refinement, coarsening or edge-flips can also change the earlier vertex counters of vertices in the updated footprint. For these cases, it is sufficient to develop heuristics for each adaptive operation that compares vertex time-coordinates before and after the adaptive operation to determine the updates to the earlier vertex counters. During the store stage if any vertex has its earlier vertex counter decremented to zero during the course of the pitch or refinement operations, it is marked as a *new* local minima and is added to the minima vertex heap of the relevant wrangler process.

It is important to reiterate that these algorithms update data in process-private memory already gathered as part of footprint construction. Updates to the earlier vertex counters reflect only changes local to the native footprint and are safe-guarded from changes outside the footprint by the vertex locks. Additionally, this algorithm allows local minima vertices to be identified and updated at the granularity of a patch; exactly the same as all the other parallel operations described in this chapter.

3.3.5 Wrangler time-heap management

When we encounter collisions from another process while locking a footprint, we need to defer the pitch about the current minimum vertex and choose another from the native process' local MVH. Since we expect some non-negligible time to elapse before the current minimum vertex becomes pitchable, we need to penalize the current vertex when choosing the next vertex. This prevents unproductive re-checking of blocked vertices before they become unlocked.

Only once a minimum vertex becomes a pitchable vertex can we build a patch over it.

To this end, the wrangler processes has two time min-heaps, the MVH and the PVH (see Table 3.2) as data members. The MVH stores the time ordered list of local time-minimum vertices that are partitioned on each wrangler. Any minima vertex over which footprint construction fails due to conflicting holds or locks are placed in the PVH. We use one of three criteria to move vertices from the PVH to MVH:

1. We successfully pitch over a vertex from the MVH,
2. We reach the capacity of the PVH,
3. The MVH becomes empty.

These criteria ensure that a non-negligible amount of time passes before a vertex is reinserted into the MVH. When any criteria is satisfied, we move all the vertices in the PVH onto the MVH as the state of holds and locks on the front can be expected to be changed. The capacity of the PVH should be a small integer greater than 1 so that failed vertices do not spend unnecessarily long in an inactive state. Algorithm 10 depicts the pseudo-code for choosing the next local minima vertex which is executed by the wrangler process at the start of footprint construction. Task-stealing — where wranglers can request additional minima vertices from over-loaded wranglers — can be implemented here if both the MVH and PVH are empty.

A well-known method to deal with collisions, the *Binary Exponential Backoff* algorithm [55], was also considered for this purpose. The BEB algorithm is commonly used for managing collisions in network traffic. In this method, once a minima vertex fails to pitch, we place it onto the PVH with a sleep interval determined by the BEB model. An exponentially increasing amount of sleep time is added after each successive collision until a maximum cap and the vertex is not considered until this time has elapsed. In practice, we find that the simple criteria proposed initially is sufficient to minimize unproductive re-checking of blocked vertices and the increased complication of the BEB model is unnecessary.

3.3.6 Data Partitioning Methods

In general, the ideal partitioning scheme should achieve the following — often competing — goals [34]:

1. Assign equal number of data entities on each subdomain,
2. Minimize communication costs between subdomains,

Algorithm 10 Choosing local time-minimum vertices from time heaps

INPUT: `successful_pitch` - Indicates the previously popped vertex has been successfully pitched

OUTPUT: `minima_vert_id` - ID of local minima vertex over which to construct patch

```
1: procedure GET MINIMA VERTEX(successful_pitch)
2:     if pending_vertex_heap.size() > maximum PVH size OR successful_pitch then
3:         Flush contents of pending_vertex_heap into minima_vertex_heap
4:         minima_vert_id ← minima_vertex_heap.pop()
5:     else if minima_vertex_heap.empty() then
6:         if !pending_vertex_heap.empty() then
7:             Flush contents of pending_vertex_heap into minima_vertex_heap
8:             minima_vert_id ← minima_vertex_heap.pop()
9:         else                                                    ▷ Task stealing
10:            Request pitchable vertices from remote wrangler ranks
11:            minima_vert_id ← NULL
12:        end if
13:    else
14:        minima_vert_id ← minima_vertex_heap.pop()
15:    end if
16: end procedure
```

3. Operate in parallel on already distributed data,

4. Re-balancing elements within subdomains after adaptive operations must be asynchronous and quick,

5. Re-balancing must be incremental *i.e.,* small changes in workload produce only small changes in the decomposition,

6. Operate on the same granularity as other parallel operations.

Traditional finite element schemes favour minimizing computation costs between subdomains since the exchanges of solution data for adjacent objects that are assigned to different subdomains, or “ghost elements”, are required during computation. For adaptive simulations, the cost of re-computing the subdomains to balance computational load is often significant. Most partitioning schemes utilize spatial coherence to minimize communication costs and any dynamic partitioning schemes respect this coherence.

Round-robin data partitioning

A traditional dynamic partitioning — as highlighted earlier in this dissertation, and when used in past attempts at developing parallel cSDG architectures (see Section 3.1.1) — is found to be ineffective due to the highly dynamic, fine-grained cSDG adaptive meshing. We therefore rely on a simple *round-robin* partitioning scheme where each front entity is assigned to the a server process using its unique, *global ID*. Given the front mesh, τ_i, at an arbitrary state of the solution algorithm and N server processes, we partition the front onto each server to form the distributed front mesh, τ_i^n, where n $= 1 \ldots N$, as is outlined in Algorithm 11. The server process onto which a front entity is mapped can be determined simply based on the global ID of the front entity. For the initial decomposition, τ_0^n, we simply assign an entity from the initial front mesh to a server if the rank of the server is equal to `dest_rank` determined by the round-robin mapping. When a new front entity is to be locally inserted onto a server process, the global ID can simply be generated locally using Algorithm 12 without the need for global communication or synchronization. In this way, dynamic re-partitioning occurs every time a new front entity is generated and is at the same granularity as the other parallel operations of our algorithm. It is evident that the round-robin partitioning scheme satisfies all but the second point in the goals for an ideal scheme discussed at the start of this section.

Algorithm 11 Pseudo-code to determine destination index for data items in round-robin scheme

INPUT: `gid` - Global ID of data element, `total_procs` - Total number of server processes over which to distribute data elements

OUTPUT: `dest_index` - The index of the owning process

```
1: procedure DETERMINE ROUND-ROBIN INDEX(gid, total_procs)
2:     dest_index ← MOD(gid, total_procs)
3: end procedure
```

Algorithm 12 Pseudo-codes to generate a global index for a new local data element

INPUT: `num_allocated` - Count of local data elements currently assigned to the process, `total_procs` - Total number of server processes over which to distribute data elements, `local_index` - The index of the local process among the total number of processes

OUTPUT: `gid` - Global ID of data element

```
1: procedure GENERATE GLOBAL ID(num_allocated, total_procs, local_index)
2:     num_allocated ← num_allocated + 1
3:     gid ← local_index + num_allocated × total_procs
4: end procedure
```

While this scheme sacrifices spatial coherence, it does allow us to devise powerful load and data balancing schemes, presented in Section 3.3.7. The drawback of the round-robin partition scheme is the large latency in gathering the required front entities from the respective server processes. Given the patch-by-patch nature of the cSDG solution scheme however, the amount of data required to be communicated is greatly reduced when compared to other numerical methods. Additionally, since all communication is performed by the server and wrangler processes, the solvers are shielded from the latency as long as the solve queues do not empty. Effectively, the asynchronous nature of the cSDG method allows us to overlap communications with computation. In this way the communication latency — which is usually a major bottleneck in scalable parallel computations — is entirely eliminated. We examine the validity of this statement in numerical studies performed in Section 3.5.

3.3.7 Probabilistic data and load balancing

There are only two distributed data structures in our architecture: one is the distributed front mesh held by server processes; the other is a comprehensive list of local time-minimum vertices of the front held by wrangler processes in the MVHs. Maintaining an even balance of these data members across the distributed system is essential for parallel scalability. A requirement for any balancing scheme is the capability of keeping up with the highly dynamic mesh adaptivity implemented by the cSDG method while minimizing communication costs needed to achieve the balance. In this section, we present scalable and cost-effective asynchronous probabilistic algorithms for both data and load balancing that operate at the granularity of individual patches.

Data balancing schemes

Our goal for balancing front data is for all server processes to hold roughly the same numbers of front vertices and front chambers at all times throughout the course of a simulation. Having abandoned the need for preserving spatial coherence in our partitioning strategy, wranglers can freely assign new vertices and chambers to any server in the distributed system. Our data balancing requirement then reduces to a *Balls-into-Bins* problem, specifically, a heavily loaded case of Balanced Allocations as presented by Berenbrink *et al.* [56]. We implement a simple scheme in which, for each insertion of a new front item, a wrangler checks the number of items of the same type stored by a small random sample of servers and assigns the new item to the least loaded server in that group. Theory presented in Section 3.5.5 predicts that algorithms of this sort attain even distributions with high probability. In practice, we find that this simple scheme performs extremely well. To reduce latency in querying sizes from servers, we utilize one-sided Remote Directory Memory Access (RDMA) MPI calls where the requesting process can directly access the required data on the remote server without active participation of the server. Algorithm 13 outlines the procedure for selecting an appropriate server rank on which to perform an insert operation.

Algorithm 13 Pseudo-code for data balancing

INPUT: `n` - Number of server ranks; `d` - Number of random ranks to probe
OUTPUT: `dest_rank` - Server rank on which to perform insert

```
 1: procedure DATA BALANCE(n, d)
 2:     min_size ← INT_MAX                              ▷ Initialize to a large positive integer
 3:     tie_break ← 0.0
 4:     dest_rank ← -1
 5:     loop over d samples
 6:         rand_rank ← random integer sample over (0, n)
 7:         size ← size of front container on server rank rand_rank
 8:         temp ← random double over $(0, 1)$                  ▷ Used to resolve ties
 9:         if size < min_size then                  ▷ Identify rank with smallest size
10:             tie_break ← temp
11:             dest_rank ← rand_rank
12:             min_size ← size
13:         else if size = min_size then
14:             if tie_break > temp then         ▷ For ties, use a random value to resolve
15:                 tie_break ← temp
16:                 dest_rank ← rand_rank
17:                 min_size ← size
18:             end if
19:         end if
20:     end loop
21: end procedure
```

Load balancing schemes

Since we only build patches over front vertices that are local minima in time, the MVHs collectively contain all current front vertices that can serve as base vertices in new footprints. Thus, pushing a new time-minimum vertex onto a wrangler's MVH effectively assigns additional computational load to that wrangler's paired solver. We develop a similar scheme for balancing computational load, outlined in Algorithm 14, but with a different criterion for defining load balance. Rather than assign equal numbers of patches to the solver tasks, our goal is for each solver to maintain *equal*, or close to equal, progress toward advancing all of the vertices in its MVH beyond the simulation's terminal time. This criterion gracefully balances load across patches with wide differences in computational costs, such as different number of cells, polynomial orders and non-linear physics models. In this case, progress is measured by the earliest time associated with any vertex in a wrangler's MVH. Wranglers with smaller earliest times lag behind others with larger times. Whenever a wrangler detects a new time-minimum vertex in an updated footprint (see Section 3.3.4), it checks the earliest MVH times of a small random sample of wranglers using RMDA MPI calls and assigns the new time-minimum vertex to the wrangler in the sample with the *largest* earliest time. This slows the leaders, allowing the laggards a chance to catch up. A further benefit of this approach is that it prevents the *implicit locking* (as described in Section 3.3.1) by maintaining a "flat" front mesh. Although this scheme falls outside the structure of standard balls into bins problems, it nonetheless performs extremely well in practice.

Algorithm 14 Pseudo-code for load balancing

INPUT: `n` - Number of wrangler ranks; `d` - Number of random ranks to probe
OUTPUT: `dest_rank` - Wrangler rank on which to perform insert

```
1:  procedure LOAD BALANCE(n, d)
2:      max_time ← -DBL_MAX                                  ▷ Initialize to a large negative double
3:      min_size ← INT_MAX                                   ▷ Initialize to a large positive integer
4:      dest_rank ← -1
5:      loop over d samples
6:          rand_rank ← random integer sample over (0, n)
7:          time ← earliest minima in time-heap on wrangler rank rand_rank
8:          size ← size of time-heap on wrangler rank rand_rank
9:          if size = 0 then                                 ▷ Trivial case
10:             dest_rank ← rand_rank
11:         else
12:             if time > max_time then                      ▷ Identify rank with largest time
13:                 max_time ← time
14:                 dest_rank ← rand_rank
15:                 min_size ← size
16:             else if time = max_time then
17:                 if size < min_size then ▷ For tie-break, identify rank with smallest size
18:                     max_time ← time
19:                     dest_rank ← rand_rank
20:                     min_size ← size
21:                 end if
22:             end if
23:         end if
24:     end loop
25: end procedure
```

3.3.8 Scalable Termination of Distributed cSDG Simulations

The termination of the cSDG simulation is non-trivial for the distributed-parallel case since all parallel processes are asynchronous and have no inherent synchronization stage. We terminate the simulation when all front vertices in each server are pitched past a final simulation time, t_f. Our goal is to shut down the distributed solution when, or soon after, all front vertex times reach, or exceed, t_f. To this end, we define an *active* vertex as follows:

Definition 3.9 (Active front vertex)**.** A vertex v of the front τ is active if:

$$\tau(v) < t_f. \tag{3.8}$$

The *server's active size* is the number of active vertices it contains. Similarly, the *node's active size* is the total number of active vertices across all servers on the node and the *system active size* is the total number of active vertices across the entire distributed system.

We say that a process is *steady* if the active size on that process remains unchanged over some interval of time, I. The number of active vertices can change due to server operations that impact the state of its local data — specifically during vertex insertion, deletion or spacetime-coordinate update. For a distributed system that comprises n distributed nodes, each with m server processes, our distributed termination algorithm designates a system-wide control process, and additional n control process corresponding to the n nodes. The node control process stores the steady status of its node and its active size. The system control process likewise store the steady status of the entire system and the system active size. The simulation is ready to terminate if the system is steady and the system active size is zero.

The termination algorithm is constructed hierarchically to ultimately update the system steady state and active size. During initialization, all node and system control processes are set as steady and the node and system active sizes are set to include all initial mesh vertices. During the simulation, each server process performs the logic outlined in Algorithm 15 whenever a vertex operation occurs that changes the state of its local data structure (see Algorithm 1). Based on the type of operation performed the server process atomically updates the shared value of node active size stored on the node control process and marks the node as unsteady, indicating at least one server on the node has updated the node active size. Interprocess communication mechanisms is used for efficient communication within the node.

Each node control process periodically checks the steady status of the node with some fixed interval, I_n. During each check, if the node is steady, the node active size is communicated to the system control process. Otherwise, the system is marked as unsteady to indicate that at least one node is still unsteady. Finally, the system control process periodically checks the steady status of the entire system with an interval I_s (where $I_s > I_n$) to determine if the simulation is ready to terminate. Communication between the node and system control processes utilize MPI RDMA communications to reduce latency. When the termination criteria are met, the termination variables in the server, wrangler and solver drivers (see Algorithms 1 - 3) are updated to allow for finalization and shut-down. The respective pseudo-code to check and update the termination quantities are presented in Algorithms 16

Algorithm 15 Pseudo-code to update termination quantities on server processes

INPUT: `op` - Operation type performed by server processes on vertex `v`
OUTPUT: `node_active_size` - Number of active vertices on node; `node_steady` - Steady status of node

```
 1: procedure UPDATE TERMINATION QUANTITIES SERVER(op)
 2:     if op = insert vertex then
 3:         node_active_size ← node_active_size + 1
 4:         node_steady ← false          ▷ Mark node unsteady due to vertex operation
 5:     else if op = delete vertex then
 6:         node_active_size ← node_active_size - 1
 7:         node_steady ← false
 8:     else if op = update vertex then
 9:         t ← v.get_time_coordinate()
10:         if t ≥ final simulation time, $t_f$ then
11:             node_active_size ← node_active_size - 1
12:             node_steady ← false
13:         end if
14:     end if
15: end procedure
```

Algorithm 16 Pseudo-code to update termination quantities on node control processes

INPUT: `node_steady` - steady status of node; `node_active_size` - Number of active vertices on node
OUTPUT: `system_steady` - steady status of distributed system; `active_size` - list of active vertices per node, maintained on system control process

```
1: procedure UPDATE TERMINATION QUANTITIES NODE(node_steady,
   node_active_size)
2:     if node_steady = false then
3:         Set system_steady ← false via MPI RDMA on system control process
4:     else                                                  ▷ Node is steady
5:         n ← get_node_id()
6:         Set active_size[n] ← node_active_size via MPI RDMA on system control
   process
7:     end if
8:     node_steady ← true                                ▷ Reset for next call
9: end procedure
```

and 17 for the node and system control processes.

Algorithm 17 Pseudo-code to update termination quantities on system control process

INPUT: `system_steady` - steady status of distributed system; `active_size` - list of active vertices per node, maintained on system control process

```
 1: procedure UPDATE TERMINATION QUANTITIES SYSTEM(system_steady, active_size)
 2:     for each node n do
 3:         system_active_size ← system_active_size + active_size[n]
 4:     end for
 5:     if system_steady = true AND system_active_size = 0 then
 6:         Broadcast end_cond ← true to all processes, see Algorithms 1 - 3,
 7:         Finalize and shut-down
 8:     end if
 9:     system_steady ← true                                  ▷ Reset for next call
10: end procedure
```

3.4 Experimental Set-up

In this section we describe the overall experimental set-up used for the development and benchmarking of the parallel–adaptive code.

3.4.1 Computing Platform

Scaling studies are performed on the Hardware Accelerated Learning (HAL) cluster [57] housed in the Innovative Systems Lab (ISL) at the National Center for Supercomputing Applications (NCSA). The cluster has 16 IBM 8335-GTH AC922 compute nodes, each with two 20-core CPUs based on the IBM Power9^{TM} architecture operating at a base frequency of 2.4 GHz for a total of *40 cores* per node and *640 total cores*. The Power 9 architecture supports 4-way simultaneous multi-threading (SMT). Each node contains a total of 256GB of physical random access memory (RAM). Internode communication is achieved by high-speed InfiniBand (IB) adapters.

3.4.2 Test Problem

The test problem considered is a rectangular domain with a preexisting crack as presented in Figure 3.7. Using symmetry we consider only the top-right region of the domain. Uniform mode 1 loading is applied to the top and bottom surfaces of the domain in the form of a step-function at time zero. The initial front mesh is produced by triangulating the domain, producing a structured gird of approximately 500 elements. The simulation is run for a fixed number of spacetime patch solves across all solver processes to ensure that the unit of work is constant across all runs. Quadratic ($p = 2$) interpolating functions are used for the solution fields, unless otherwise mentioned.

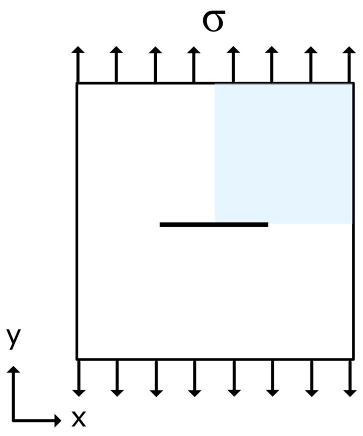


Figure 3.7: Schematic of test problem used to verify and benchmark the parallel scalability of the method. Shaded quadrant is explicitly considered with a predefined crack loaded under mode 1 uniform loading.

3.4.3 Simulation Cost

The implemented method is verified using the one-field elastodynamics equations (*cf.* Section 2.3) for the case of two spatial dimensions × time. The degrees of freedom per spacetime element n can be determined as

$$n = 3\frac{(p+d)!}{p!d!}, \tag{3.9}$$

where p is the polynomial order of the function interpolating the solution field and d is the total dimensions of the problem ($d = 3$ for two spatial dimensions). The factor 3 is to account for the three components of a single vector field. Since solutions are usually computed over patches of elements, we have the total number of degrees of freedom per patch as

$$N = Kn, \tag{3.10}$$

where K is the number of elements per patch. Generally, K is a function of the degree of a vertex in the front. We therefore find that the complexity is linear in the number of degrees of freedom per patch.

3.4.4 Strong versus Weak Scaling Efficiency

Parallel scalability is an indication of how efficiently any parallel code takes advantage of increasing number of parallel processing elements, or cores in this case. This can be measured by the *strong scaling efficiency* and is defined as

$$\varepsilon_s = \frac{T_1}{N \times T_N} \times 100\%, \tag{3.11}$$

where T_1 is the time needed to complete a task using 1 processing element and T_N is the time needed to complete the same task using N processing elements. For strong scaling, the problem size stays fixed. For a perfectly scaling problem, ε_s is 100 %. In general, it is hard to get good scaling for large number of cores as the communication and data-access costs increase in proportion to the number of cores.

For a large distributed system, *weak scaling efficiency* is typically used to measure parallel performance. Unlike the case of strong scaling efficiency, the problem size is scaled along with the number of parallel processing elements when computing the weak scaling efficiency. This results in near-constant work for each parallel core added. Weak scaling efficiency is defined as:

$$\varepsilon_w = \frac{T_1}{T_N} \times 100\%, \tag{3.12}$$

where T_1 is defined as above and T_N is the time needed to complete N of the same tasks using N processing elements. For $\varepsilon_w = 100\%$ we would solve a problem N times larger using N parallel processing elements in the same time it would take to solve the original problem in serial.

Before we can measure the scaling efficiency of our program, the definition of the baseline or serial performance, T_1 must be discussed. Since we use three distinct and asynchronous processes (the wrangler, server and solver) in the parallel–adaptive code there is some ambiguity when defining the baseline configuration. One option is to run all processes on a single thread and sequentially make progress over each state machine. The other option is to define the baseline configuration as a single solver process, along with one server and one wrangler processes running asynchronously on multiple threads. In this way, by increasing the number of solver processes, we are effectively adding additional parallel processing elements as long as the supporting wrangler and server processes are sufficient to supply footprints to the added solvers. For the results presented here, we use the second option as we argue that limiting the processes to a single thread would unfairly hinder the baseline case. Therefore, we redefine T_1 as the time needed to complete a task using 1 *solver process* and T_N as the time needed to complete the task using N *solver processes*. In order to compute weak efficiency in the cSDG code, we simply duplicate the initial front mesh of the problem described in Section 3.4.2 for each distributed node in the simulation. In this fashion, the number of front vertices — over which wranglers can construct footprints to provide work for the solvers — are scaled along with the added solver processes.

3.4.5 Data Collection Window for an Adaptive Code

The adaptive code has periods — especially when the simulation has just been launched on a coarse initial mesh, and at the end of the simulation when the number of active front vertices (*cf.* Section 3.3.8) are shrinking — where the overall number of pitchable vertices are much smaller than other stages of the simulation. In order to accurately measure scaling in the adaptive code, we need to define a robust metric that is insensitive to stages where the available number of pitchable vertices (or "work" in the general sense) fluctuates due to adaptive operations. To this end, we define a *data collection window* based on the wall-clock simulation time during which we can expect the number of pitchable vertices on the front to be sufficient for all, or a large proportion of, solvers to be kept busy. In other words, we need to ensure that there are close to zero solve-queue stalls during data collection. To trigger the start of the data collection window the following criteria is periodically tested on *each* wrangler process:

1. Is the size of the wrangler MVH greater than some fixed number, α?
2. Has the solver process paired to the wrangler solved more than some fixed number of patches, β?
3. Is the solve queue full?

Once these criteria are met on a particular wrangler, we communicate this to a global monitoring process that is in charge of starting the data collection window across the entire distributed system when all wranglers report the same. To trigger the end of the data collection window we take a simpler approach where we mark each wrangler as having completed collection of data once a fixed number of patches, γ, have been solved by its paired solver. The choice of the integer parameters α, β and γ used to start and stop the collection window are problem-dependent and must be tuned carefully to ensure that once collection has started, there are close to zero solve queue stalls.

3.5 Numerical Results and Optimizations

This section presents numerical results for the scaling of the distributed parallel–adaptive cSDG method. We benchmark the code as outlined in the preceding section. We present various optimizations and numerical results for the parallel–adaptive code to obtain optimal scaling efficiency. Some of these optimizations are machine dependent and may need to be carefully tuned to yield optimal results for each system architecture.

3.5.1 Process Distribution and Affinity Mapping

The most important consideration for the scalability of the parallel–adaptive code are solve queue stalls — periods where solver processes are idle due to lack of work from the upstream wrangler. Since the solvers are designed to be embarrassingly parallel, they are capable of continuously performing useful work as long their solve queue contains a new task every time one is completed. The increase in latency due to the spatially incoherent round-robin distribution scheme can only be hidden when there are no solve queue stalls.

The focus then shifts to the upstream wrangler (and by extension, server) processes which are responsible for feeding the solver with footprints. We need to ensure that wranglers are gathering front entities and constructing footprints at a rate equal to or greater than the rate at which they are consumed by the solvers. Additionally, servers need to respond to requests from wranglers at a satisfactory rate.

Determining an optimal process distribution

We can consider the three process types in two categories: those that perform useful computations, namely the solver process and those that handle the traversal and maintenance of the distributed front, namely the wrangler and solver processes. Given a distributed system with fixed computational resources, we need to determine a viable distribution of these two categories of processes for greatest parallel performance. One distribution of processes would be to devote nearly all system resources for the former category and minimize resources for the latter thus devoting the majority of the system to useful number-crunching. However such a distribution would be unfeasible due to solve queue stalls stemming from a lack of resources on the upstream wrangler and server processes. Thus, the optimal distribution of processes would aim to minimize the proportion of system resources used for wrangler and server processes such that the solver processes experience no solve queue stalls. In this way, the latency of the partitioning scheme is entirely hidden from the solvers, one of two requirements for near-perfect scalability identified in Section 3.2.2.

Process affinity mapping

Affinity mapping is the explicit binding of execution processes to specific hardware resources. This may be on the granularity of individual cores or even hardware threads. By default, processes are not tied to a particular resource but rather, the available resources are divided in some way by the *scheduler*. This is usually done in a way that ensures fairness among the current tasks such that no task will be denied access to system resources for extended periods of time. The scheduler "swaps" the currently executing task for another at regular intervals. This is done at some expense as the state of the currently executed task must be saved and the new task must be loaded. The fast cache memory may be evacuated to make space for the new task and data may need to be fetched from the slower main memory at greater frequency due to thread swapping. This phenomenon is exacerbated when the hardware resources are not sufficient for the number of tasks. Thread affinity mapping is used to override the default scheduler when the performance properties of the program are well known. By doing so, each execution thread is granted exclusive ownership of hardware resources, improving performance.

Another feature of modern-day multi-threaded processors is the *non-uniform memory access* (NUMA) architecture where memory is physically distributed but logically shared within a node. Due to this distribution, memory access time depends on the location of memory relative to the processor. The affinity mapping should take into account the NUMA architecture of the system and store data in such a way that the memory access times are minimized.

We employ affinity mapping for all process types to improve the performance of the parallel–adaptive code. The server and wrangler processes are light-weight when compared to the solver process. Additionally, due to the latency in communication across the distributed network, we expect periods when server and wrangler processes are in an idle state while awaiting the response of a message. Based on this usage pattern, we choose to map multiple servers and wranglers onto the same physical core to take advantage of SMT available on all modern processors. When one process enters an idle state, other processes mapped to the same core can be swapped in to utilize the core's resources. On the other hand, the solver processes are expected to be constantly busy with computations for the entirety of the simulation. We therefore map only one solver process per physical core to ensure that all resources on that core are available to the solver.

A viable distribution and affinity map for the HAL cluster

We aim to determine a viable distribution of processes along with an optimal affinity map on the HAL system such that we have zero queue stalls and have an acceptable patch solve rate. At the moment, this tuning is performed iteratively through a process of trial and error until we satisfy the above goals. In the future, an auto-tuning step could be devised to automatically determine a viable distribution for when exploring a new system architecture.

A NUMA node on the HAL cluster consists of 20 CPU cores, each supporting up to 4-way SMT. We experiment with a number of different process distributions and affinity maps while closely monitoring the queue stalls and patch solve rate. Affinity mapping is achieved using the Portable Hardware Locality (hwloc) utility [58] which provides a light-weight and portable way to identify hardware topology and bind processes to CPU and memory regions. We define the *percentage of solver stalls* as the percentage of times a solver process encounters an empty solve queue over the total number of times the solver tries to grab a task from the solve queue. We tally only the first occurrence of a stall to avoid counting many unsuccessful re-checks of the queue in a small interval. The results for our tuning on the HAL system are presented in Table 3.4. Since we pair wrangler and solver processes through the solve and store queues in our current architecture, the number of these process are equal. The tuning runs were performed on a single node and we monitor average statistics over all solver processes. The patch solve rate is determined as the total number of patches solved over the entire node during the timing window.

Table 3.4: Tuning results for different process distribution and affinity mapping configurations over 1 NUMA domain on the HAL cluster. Experiments were performed for $p = 2$ and two spatial dimensions.

Configuration			**% of Solver Stalls**	**Overall patch solve rate (pathes/s)**
Num. solvers	**Num. servers**	**Affinity Mapping**		
20	2	None	21.2	172
18	2	None	15.6	212
18	2	Servers and wranglers mapped to one core each; solvers mapped over remaining cores	10.1	249
18	4	Same as above	2.1	316
18	8	Same as above	0	387
16	8	Servers and wranglers mapped over two cores each; solvers mapped over remaining cores	0	336

The results indicate that a viable configuration is one with 8 server processes, along with 18 solver and wrangler processes which results in zero queue stalls and the greatest patch solve rate. Any more solvers causes excessive queue stalls due to lack of resources for the wrangler and server processes. We also find that the optimal mapping scheme involves pinning all server processes onto a single dedicated "server core" and all wranglers onto a dedicated "wrangler core". As discussed above, these processes are comparatively light-weight and can tolerate being swapped-out during idle states. The 4-way SMT available on HAL allows us to isolate all server and wrangler processes onto just two physical cores and map one solver process onto each of the remaining physical cores. Thus, the final configuration allows for 90% of total system resources to be dedicated to perform the useful computations. Distributing the wrangler and server processes over more physical cores yields no benefits since the production of footprints was sufficient in the previous configuration. It is important to note that this distribution and mapping configuration is dependent on polynomial order and dimension of the underlying problem since these would greatly affect the consumption of footprints by the solver processes.

3.5.2 Inter-node Communication Pattern and Impact on Solvers

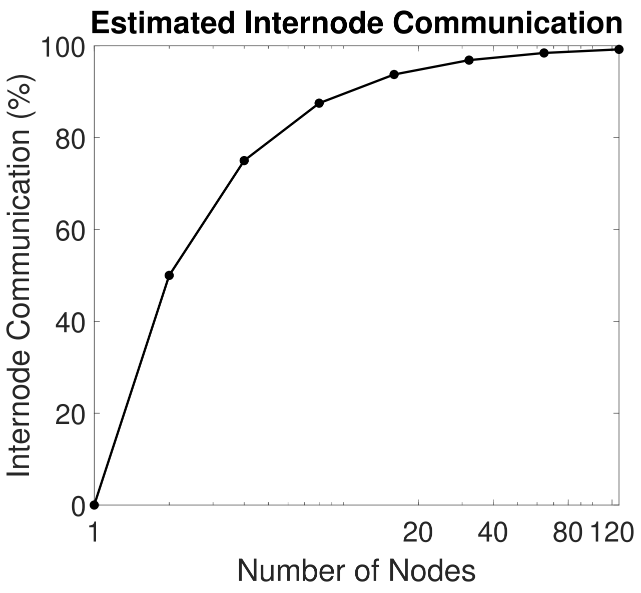
The ability to manage both load and data while keeping up with fine grained adaptive operations is a major feature of the parallel–adaptive code. As previously discussed, by abandoning the DDM we utilize a round-robin partitioning scheme that is not aware of spatial coherence. This means that the communication costs are necessarily larger when gathering and scattering front data. In this section, we examine the communication pattern between the server and wrangler processes and investigate its impact on code performance.

Estimated internode communication

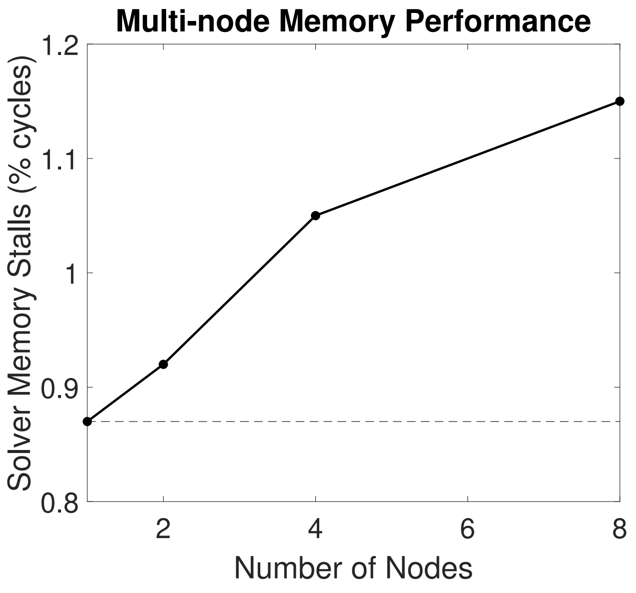
Given a distributed front partitioned using the round-robin scheme outlined in Section 3.3.6, we can expect that for each piece of data required to be gathered/scattered, there is a probability of $p = (N-1)/N$ that the data resides on another node, where N is the total number of distributed nodes in the system. Thus, for very large systems, we expect $p \to 1$. Figure 3.8a depicts this estimated internode communication where the asymptotic behavior is clearly seen.

Solver memory stalls

When benchmarking the code as described, we find that the solver processes experience *memory stalls* that increase as we increase the number of overall solver processes. Memory stalls are events during which a process is stalled and unable to do useful work waiting for a memory access. This commonly occurs due to a slow access to main memory for needed data. In practice, these are measured as percentage of total CPU clock cycles spent in a stalled



(a)



(b)

Figure 3.8: Effect of inter-node communication pattern on memory stalls experienced by solver processes. (a) Semi-log plot of the estimated internode communication as a function of the number of distributed nodes for the round-robin partitioning scheme; (b) Percentage of total cycles stalled due to memory accesses by the solver processes as a function of the number of distributed nodes.

state. We use the TAU profiler [59] to measure various performance metrics related to the execution of the parallel code. To measure the percentage of cycles stalled, we utilize TAU API calls to monitor one representative solver process during the simulation, yielding the plot as presented in Figure 3.8b.

Since the solver processes are designed to be embarrassingly-parallel and do not rely on data generated from other processes during the solution stage, we expect the percentage of stalls for the solver to be close to constant. However, this is clearly not the case as evidenced by Fig. 3.8b where memory stalls increase as we increase the number of distributed nodes. In fact, the overall increase in the percentage of cycles stalled follows a similar trend to that of the estimated inter-node communication *cf.* Fig. 3.8a. This finding — along with the fact that the major additional cost of adding more nodes is in the additional overhead of the MPI sub-system to process messages — leads us to consider optimizations to isolate the solver process from the interference of the MPI processes presented in Section 3.5.3.

Buffered MPI Communications

To further reduce memory stalls experienced by the solver, we investigate utilizing the buffered version of the MPI non-blocking send command, `MPI_Ibsend`, wherein the user specifies a buffer from which MPI copies the data to send. By explicitly defining the buffer in the user code instead of defaulting this choice to the MPI implementation, we achieve a modest reduction in memory stalls as depicted in Figure 3.9.

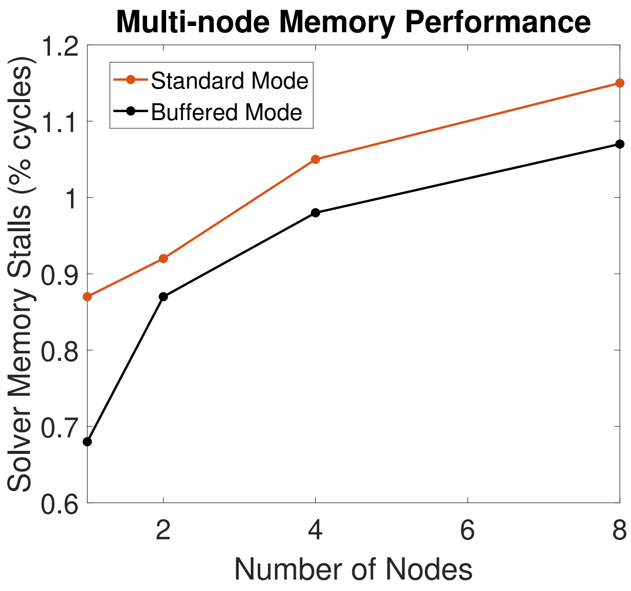


Figure 3.9: Percentage of total cycles stalled due to memory accesses for the buffered and standard MPI send commands.

3.5.3 Solver Process Isolation

As discussed in Section 3.5.1, as long as we have identified a distribution of processes that generate zero solve queue stalls, we should expect near-perfect parallel scaling. However, in practice, we find that this is far from the case. We plot, as a baseline, the weak scaling efficiency of the code with the optimizations presented thus far in Figure 3.10. We find that even though there are no solve queue stalls across the distributed system during the data collection window (*cf.* Section 3.4.5), we see a drop to $\sim 50\%$ weak scaling efficiency at 16 distributed nodes.

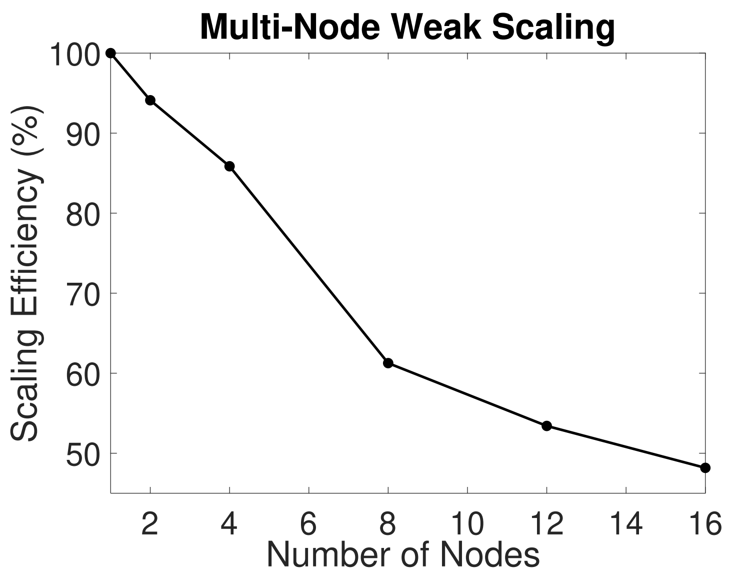


Figure 3.10: Weak scaling efficiency of distributed–parallel code without optimizations to isolate solver processes.

Solver isolation through process spawning

We hypothesize one of the main causes of this drop in scaling efficiency to be the increasing number of memory stalls in solver processes due to the lack of isolation from the MPI sub-system identified in the preceding section. When the paired wrangler performs any MPI-related task, we hypothesize that these operations effect the memory performance of the solvers. This effect is exacerbated when we increase the percentage of internode messages by adding additional nodes. In an effort to completely isolate the solver processes, which do not have any need to communicate over the distributed system, we spawn the solvers as completely independent process rather than independent threads running alongside wrangler processes as considered thus far. Communication between the now distinct wrangler and solver process must now utilize *interprocess* mechanisms since each process no longer shares a common memory address space. We use the Boost C++ interprocess library to accomplish this with minimal additional implementation effort. This library utilizes system shared memory, which is typically the fastest form of interprocess communication, to provide an area of memory that is shared between processes. Specifically, we utilize the Boost single-producer/single-consumer (SPSC) queue data structure to implement interprocess versions of the solve and store queues (which, by design, are the only places where solvers and wranglers exchange data). The SPSC queue is a FIFO queue where push and pop operations are entirely lock and wait free under the constraint of only a single consumer and single producer of data to and from the queue. By separating the wrangler and solver processes, we see a substantial reduction in percentage of stalled cycles as plotted in Figure 3.11 and a modest improvement in the weak scaling efficiency.

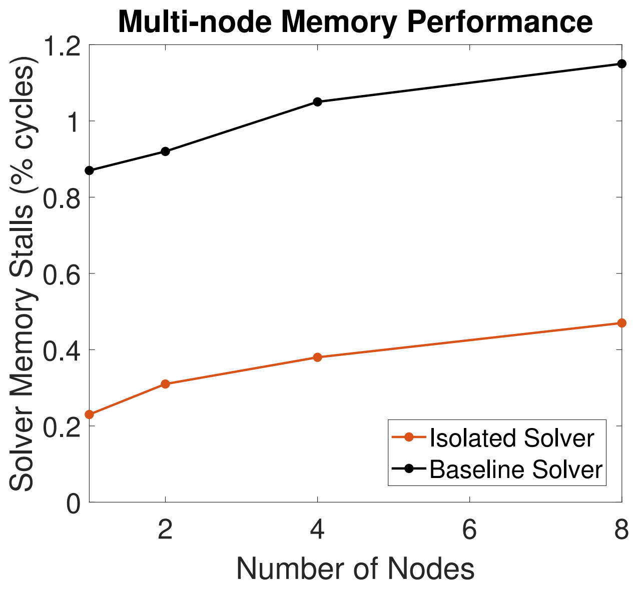


Figure 3.11: Percentage of total cycles stalled due to memory accesses for the baseline and isolated solver configurations.

3.5.4 Avoiding Dynamic Memory Allocations

Another significant source of loss of scaling efficiency is during dynamic memory allocations where multiple parallel processors are dynamically allocating and de-allocating memory (using `malloc` and `free` commands in C++) during the lifetime of the simulation. These operations utilize the system heap memory [3] which is a resource shared between multiple processes. In order to achieve thread-safety, the heap is synchronized with a mutex which must be acquired

[3]The heap is the portion of physical memory where dynamically allocated memory resides. Heap memory is allocated explicitly and will be accessible until it is explicitly freed.

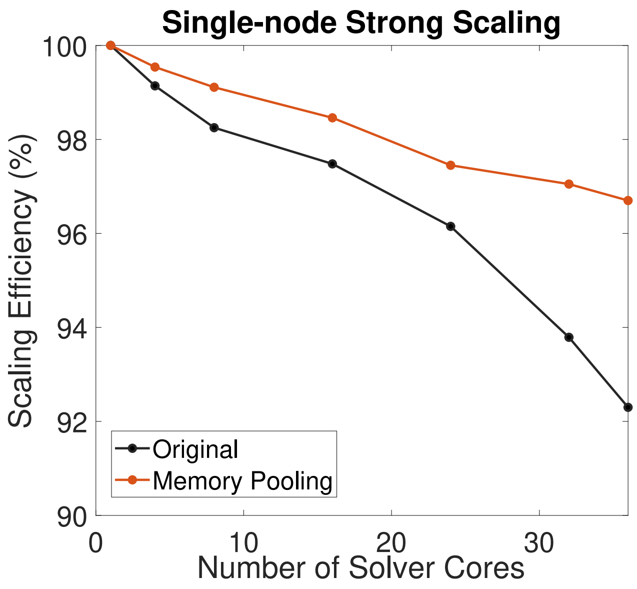
before a process can allocate dynamic memory. Contention to access the global heap adds an expensive overhead in parallel programs as processes spend a great deal time waiting on the mutex, rather than doing useful work. Additionally, dynamic memory allocations lead to further issues such as *false sharing*, where multiple processes share data in the same cache line causing poor memory performance, and *blowup*, where allocated memory has a tendency to become unbounded for certain patterns of operations.

These issues motivate the removal of all forms of dynamic memory allocation in the parallel code where possible, instead utilizing pre-allocated and pre-sized data structures created at code initialization that infrequently need to be resized during computation. We utilize three strategies to reduce dynamic memory allocations in our baseline code:

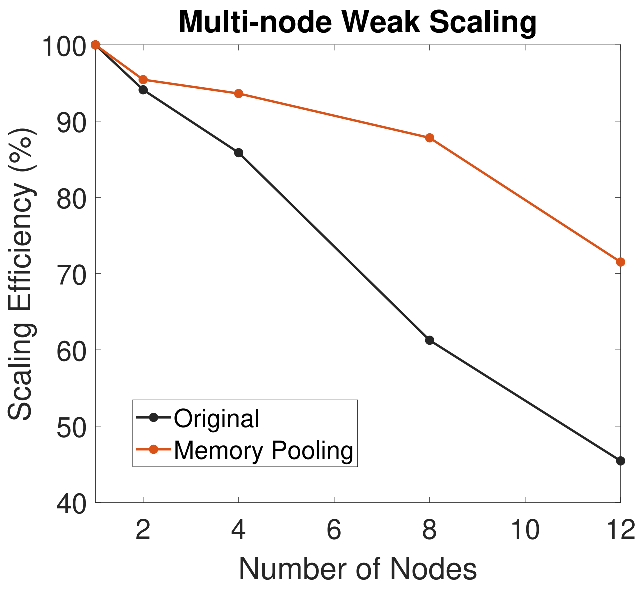
The first involves the creation of a templated, pre-allocated vector-like container called a *pooled vector*. Each pooled vector is pre-allocated using a generously chosen initial size with default values for the stored data. Augmenting the vector is a list of free indices marking the entries of the vector that are available for use in storage. When we insert a new entry onto the vector we choose a free index and over-write the currently stored data at that index. The index is then removed from the free index list. Conversely, when we choose to delete an existing entity, we simply add the index back to the list of free indices. If all indices are consumed, we are forced to dynamically allocate new objects to accommodate additional insert requests. We choose to double the size of the pooled vector each time we perform such a resize to avoid multiple successive memory allocations. This container is used to store the data elements for the distributed front, the MPI buffer and message objects, as well as footprint and patch data structures on the wranglers and server processes.

The second strategy is to use the Boost C++ pool allocator library. This library creates a large pool of shared memory in physical memory at initialization and partitions it into many smaller segments with the same size. Each time the user code requests a new object to be created using a `malloc`-style command, the library accesses the next free segment and assigns memory from that segment. The segment is then marked as used. When the memory segment is no longer required, the library releases the segment back into the pool. The pooled allocator is especially efficient during the allocation and deallocation of many small objects. We utilize pool allocators for the storage of the many intermediate vectors and matrices formed during the finite element assembly and solution stage on solver processes.

Finally, the third strategy utilizes the hoard memory allocator library [60], a fast, scalable memory allocator that is designed to address the aforementioned issues with dynamic memory allocations. This run-time library automatically replaces any remaining calls to `malloc` and `free` in the baseline code with optimized versions that are lock-free and address the issues of false-sharing and blowup.



(a)



(b)

Figure 3.12: Scaling studies illustrating the improvements when utilizing memory pooling strategies. (a) Strong scaling efficiency for a single node; (b) weak scaling efficiency for up to 12 distributed nodes.

These strategies are used extensively in the parallel–adaptive code to eliminate dynamic memory allocations where possible. We present the results as compared to the baseline code in Figure 3.12 for both the cases of strong and weak scaling efficiency. We achieve an impressive improvement of $\sim 25\%$ weak scaling efficiency just from treating the dynamic memory allocations in the baseline code.

3.5.5 Parameter selection for Balancing Algorithms

An integral part of the distributed architecture is the round-robin partitioning scheme and the resulting ability to balance both load and data at the same granularity as the patch solves. Here we examine and determine optimal parameters for the balls-into-bins algorithm outlined in Section 3.3.7.

Suppose we have n processes (or bins) onto which we need to place m finite number of pieces of data. The processes can communicate to each other with some non-negligible cost. Our goal is to ensure that each of the n processes maintain equal, or close to equal, number of data items while minimizing the communication costs required to maintain this balance. The problem can be cast into the well known *Greedy(d)* randomized algorithm [61] for load balancing. When tasked to insert a new ball into the n bins, the algorithm chooses $d \geq 2$ random bins for each ball and inserts the ball into the bin with the least load, or number of data items. The m balls are inserted sequentially in this fashion by performing d samples at each stage. If several bins have the same maximum load, the ball is arbitrarily placed into one of them. The question then becomes — what is the smallest value of d such that the difference between the maximum load over all bins and the average number of balls per bin (*i.e.,* m/n) is minimized. For the heavily loaded case where $m \gg n$, we have the following useful result:

Theorem 3.1 (Heavy Loaded Greedy(d) [56])**.** If m balls are allocated into n bins using Greedy(d) with $d \geq 2$ such that $m \gg n$, then with high probability the largest number of balls in any bin is at most

$$\frac{m}{n} + \frac{ln\, ln\, n}{ln\, d} + O(1). \tag{3.13}$$

This theorem can be used to choose a value for the parameter d that gives us an acceptably small deviations from the mean number of balls in any bin. Suppose we wish to make the deviation a constant, *i.e.,* for some constant integer $k \geq 1$ we require:

$$\frac{ln\, ln\, n}{ln\, d} \leq k \Rightarrow n \leq exp(d^k).$$

Thus, from Theorem 3.1, we have with high probability that the largest number of balls in any bin is at most

$$\frac{m}{n} + k + O(1) = \frac{m}{n} + O(1)$$

as long as the number of bins satisfies $n \leq exp(d^k)$.

For example, if we set $k = 3$ and $d = 2$ we find that for $n \leq exp(d^k) \approx 8000$ bins with high probability we will find that the bin with the maximum load will be at most a fixed constant k more than the mean number of balls per bin. Setting the number of random samples instead to $d = 4$, we find that $n \approx 10^{35}$. This simple analysis implies even for very large number of bins we can choose d as small as 2 and guarantee with high probability the maximum deviation from the mean number of balls per bin is limited to a fixed constant.

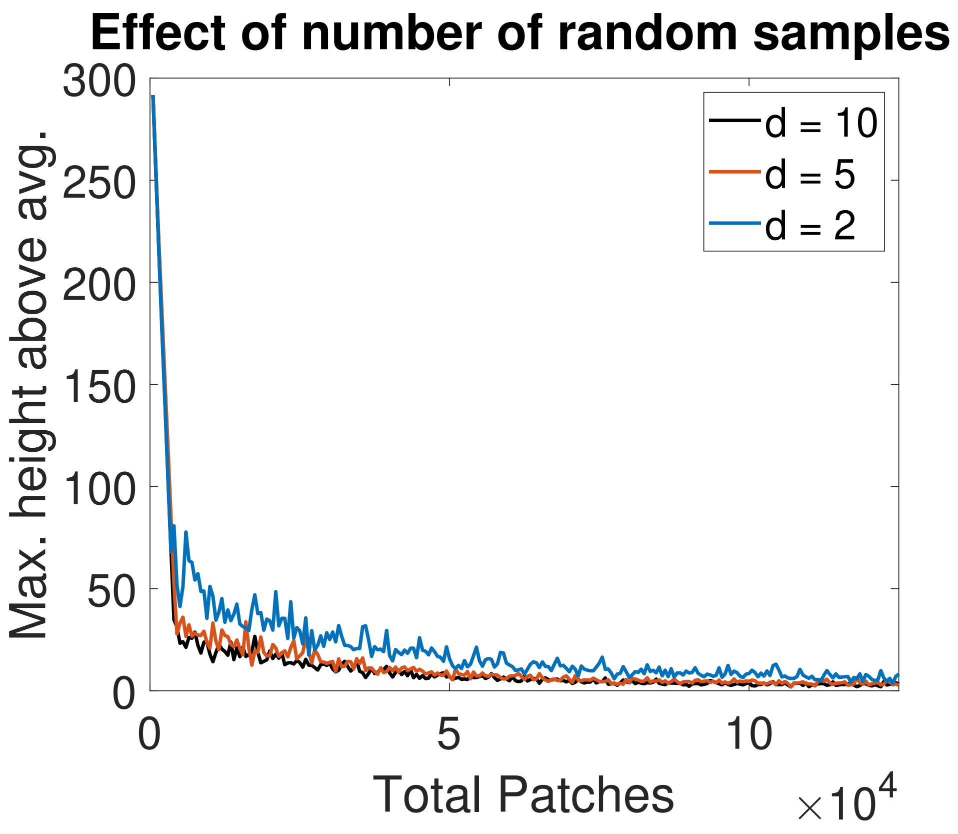


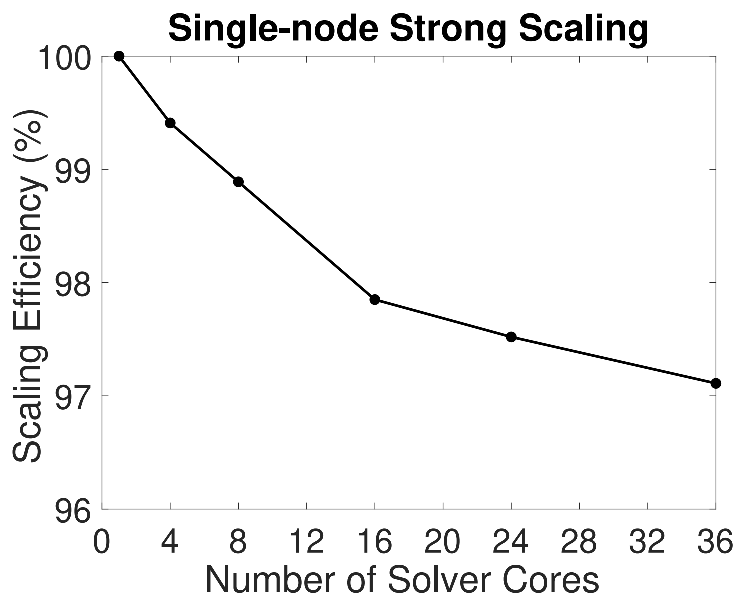
Figure 3.13: Maximum height over average number of balls per bin versus total number of patches solved for varying values of random samples, d, used in the Greedy(d) algorithm. Starting with an unfair initial distribution of 300 balls into just one of 36 total bins, the balancing algorithm works to reduce the maximum height over multiple iterations (or patch solves in the case of the cSDG method).

To verify the performance of the Greedy(d) algorithm in practice, we first define the *maximum height above average* as the difference between the maximum load over the entire distributed system and the average number of balls per bin. We then set-up a contrived example where initially all balls are placed in just one bin. We then allow the balancing algorithm to work to balance the balls across the bins as new balls are added, for different values of d. We plot the maximum height above average over multiple iterations (or patch solves in the case of the cSDG method) in Figure 3.13 for $d = 2, 5$, and 10. We find that the Greedy(d) algorithm aggressively reduces the initial imbalance over multiple iterations and is capable of restoring balance even in this contrived case. For our scaling studies in this section, we set $d = 5$ as this is more than sufficient to maintain load and data balance as demonstrated in this section.

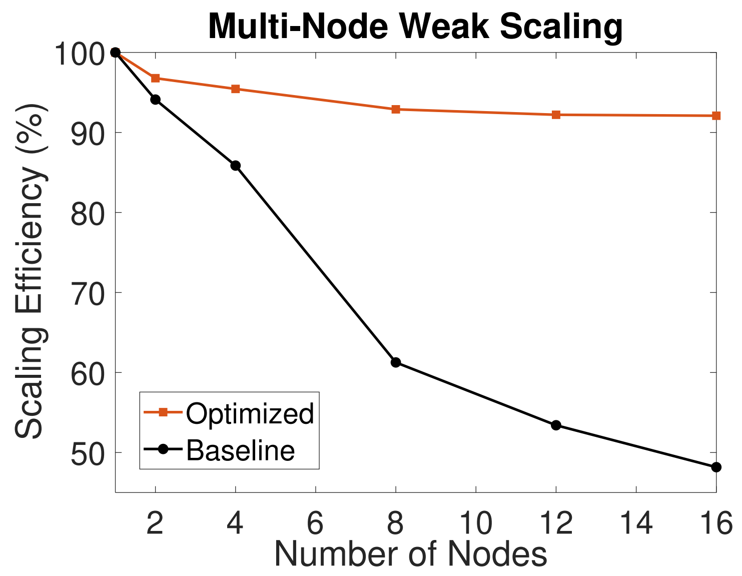
3.5.6 Scaling Results for the Parallel–adaptive Architecture

Based on all the optimizations presented thus far, we benchmark the parallel–adaptive code to determine its scalability. We report both the strong and weak scaling efficiency as computed in Section 3.4.4. The data collection window is started in line with Section 3.4.5 and terminated once each solver has completed $10,000$ patch solves. We ensure that there are no solve queue stalls for the entire data collection window. The scaling tests were performed on the two-dimensional elastodynamics problem with polynomial order $p = 2$.

We first present the strong scaling efficiency of the code for up to 36 solver cores in Figure 3.14(a) where we find that the efficiency is $\sim 97\%$ for the fully loaded node. For the weak scaling efficiency presented in Figure 3.14(b) we see that the optimizations improved performance to $\sim 92\%$ for the fully loaded distributed system. Additionally, we find that the degradation in weak scaling efficiency is leveling off much faster for the optimized case, which is a promising result when applying the code to a larger system. Addressing the drop in scaling efficiency remaining in the optimized code remains an open issue. We have identified additional data-structures in the solver code where dynamic memory allocations still occur and have not been treated by memory pooling strategies. As seen in Section 3.5.4, elimination of such dynamic memory allocations yields considerable improvement to scaling efficiency. Additionally, memory and cache performance tends to degrade as additional parallel processes are added to the distributed system. A more extensive study of cache performance has the potential to greatly improve parallel performance. Finally, we identify CPU shielding techniques as a possible way to completely isolate the solvers from background system and MPI processes. Nevertheless, these results represent the first time the fully adaptive cSDG code is demonstrated to scale favorably over a large distributed cluster.



(a)



(b)

Figure 3.14: Scaling studies illustrating the performance of the parallel–adaptive code. (a) Strong scaling efficiency for a single node; (b) weak scaling efficiency for up to 16 distributed nodes comparing the differences between the baseline solver (*cf.* Fig. 3.10) and the optimized architecture.

3.6 Chapter Summary

In this chapter we have described the overall task-based parallel–adaptive cSDG architecture and highlighted its scalability on a large distributed cluster. This is the first truly parallel–adaptive adaptation of the method capable of maintaining load and data balance across a distributed system, leading to substantial speed-up over the previous attempts. The architecture comprises three asynchronous processes: the footprint wrangler clients, distributed front servers and embarrassingly parallel patch solvers. The wrangler clients perform gather/scatter operations related to the distributed front mesh and prepare footprints that are then used to generate and solve patches by the solver processes. Front mesh server processes store, update, and respond to requests for the distributed front mesh data structure from wranglers via the MPI framework. The final architecture is asynchronous, latency-tolerant, and virtually lock and barrier free. In order to realize this, we developed many novel distributed algorithms that share the same granularity as patch construction and solution operations. Key to this is the round-robin front partitioning scheme that forgoes the spatial coherence of the traditional DDM. Instead, we detail probabilistic balancing schemes that allowed us to dynamically balance load and data across the distributed system at a rate on par with the highly dynamic mesh adaptive schemes. We further described the other algorithms implemented for scalability and correctness of the distributed code such as the lock and hold, earlier vertex counter and scalable termination algorithms. Finally, various optimizations are explored to ensure the scalability of the code when tested on the NCSA's HAL cluster. This involved determining a viable distribution of the aforementioned process types and their affinity map on the distributed cluster. Additionally, the isolation of the solver processes by removing unnecessary dynamic memory allocations and spawning the solver as separate processes from their paired wrangler yielded large improvements to parallel scalability.

Chapter 4

Spacetime Method for Direct Numerical Simulation of Multiscale Earthquake Dynamics

4.1 Introduction

Earthquakes are multiscale phenomena in both space and time. A single seismic event with millimeter-scale rupture process zones can extend over fault systems that are hundreds of kilometers in length [62, 63]. This implies a maximum-to-minimum ratio of physical length scales on the order of 10^8. Supershear ruptures can generate even more extreme ratios. Typical wave speeds and rupture-tip velocities combine with these length scales to generate similarly extreme temporal-scale ratios. Despite recent advances in high-performance computing technology, state-of-the art seismic codes are still unable to bridge the physically relevant ranges of spatio-temporal scales.

Most seismic codes use synchronous, time-marching solvers that impose uniform time-step sizes controlled by worst-case conditions, either globally or by region. Igel *et al.* [64] and Poursartip, Fathi, & Tassoulas [65] review high-performance synchronous seismic codes. Most of these are non-adaptive and rely on domain truncation or static graded meshes to reduce computational expense; *cf.* [66, 67, 68, 69]. Seismic models that support adaptive mesh refinement, such as [70, 71], are less common and refinement is only allowed after completion of one or more time steps. Despite the promise of higher-order accuracy, the use of high-order basis functions in finite element models typically reduces computational efficiency. Therefore, most seismic codes adopt low-order spatial discretizations, resulting in lower-order accuracy, and do not support p and hp-adaptive solution schemes.

These limitations make it difficult to resolve the full range of physically relevant length and time scales, and in the case of adaptive codes, to deliver the level of dynamic remeshing required to capture fast-moving wave fronts and rupture-tip process zones. This commonly forces seismic analysts to adopt unrealistic values of physical parameters or to accept under-resolved solutions, either of which undermines the reliability of simulation results.

Local time stepping (LTS) schemes [72, 73] allow time-step sizes to vary across the computational domain, with smaller time-steps in regions with greater mesh refinement or higher wave speeds. LTS methods improve computational efficiency relative to methods with globally uniform time-step sizes.

Unstructured discretizations, such as the discontinuous Galerkin (DG) method, allow for h-adaptive non-conforming meshes which offers the power to resolve complex fault geometries and greatly simplifies mesh generation for geophysical problems. Moreover, it greatly facilitates dynamic solution adaptivity, for example to track propagating wavefronts or earthquake rupture dynamics. DG schemes have been utilized extensively in the literature [74, 75, 76, 77, 78] due to these benefits.

In this chapter, we propose the cSDG method [1, 20] as a powerful solution scheme for extreme multiscale problems in seismology. The cSDG method features unconditional stability, conservation of linear and and angular momentum over every spacetime cell, linear computational complexity, and support for arbitrarily high-order elements. However, its most compelling feature is the asynchronous solution scheme that closely couples adaptive, unstructured spacetime mesh generation with localized spacetime discontinuous Galerkin solutions. In contrast to synchronous time marching, this structure enables nonuniform, asynchronous time advancement as well as an extremely dynamic form of spacetime adaptive mesh refinement. Additionally, the cSDG formulation supports arbitrarily high-order basis functions that admit discontinuities across all spacetime inter-element boundaries where weakly-enforced Riemann jump conditions preserve the governing system's characteristic structure. We use specialized Riemann solutions for separation, contact–stick, and frictional sliding to model fault physics. These features combine to deliver high-resolution cSDG solutions capable of capturing the fine details of multiscale earthquake-rupture physics.

4.2 Continuum Formulation

4.2.1 Spatial and Spacetime Analysis Domains

Consider an open *spatial domain* $\omega \subset \mathbb{E}^d$ with exterior boundary, $\partial\omega$, and an embedded fault network, $\gamma : \gamma \cap \omega = \emptyset$. We seek a solution on the *spacetime domain*, $\Omega := \omega \times \mathcal{I} \subset \mathbb{E}^d \times \mathbb{R}$,

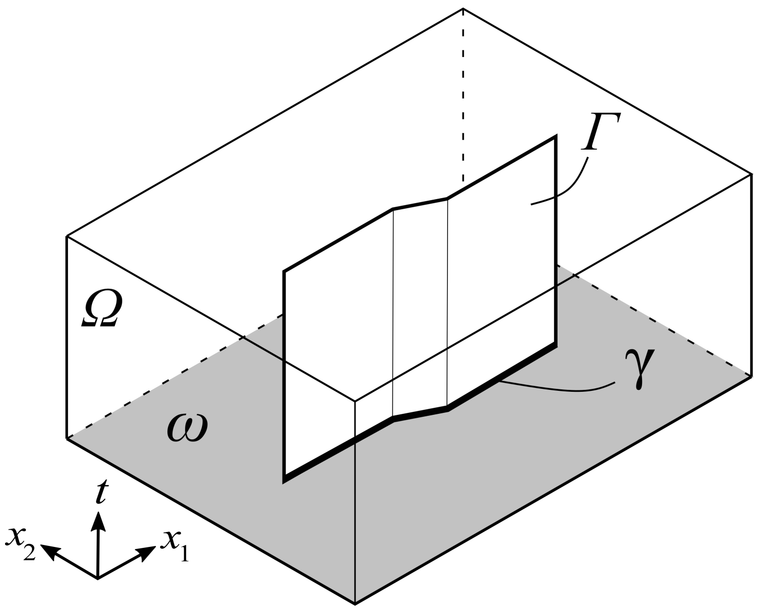


Figure 4.1: Spatial domain, $\omega \subset \mathbb{E}^2$, with embedded fault, γ, and spacetime analysis domain, Ω, with fault trajectory, Γ.

in which $\mathcal{I} := (0, T)$ where 0 and T are the initial and terminal times for the simulation. The system of oriented d-manifolds, $\Gamma := \gamma \times \mathcal{I} \subset \mathbb{E}^d \times \mathbb{R}$, is called the *fault trajectory.* Figure 4.1 depicts the two-dimensional spatial domains, ω and γ, and the corresponding spacetime domains, Ω and Γ. We use + and − superscripts to label the fault-face trajectories coincident with Γ, such that Γ^+ has the same orientation as Γ.[1]

We partition the boundary of Ω as $\partial\Omega = \overline{\partial\Omega_\mathrm{I} \cup \partial\Omega_\mathrm{B} \cup \partial\Omega_\Gamma \cup \partial\Omega_\mathrm{T}}$ in which $\partial\Omega_\mathrm{I}$, $\partial\Omega_\mathrm{B}$, and $\partial\Omega_\Gamma$ are open segments of $\partial\Omega$ on which initial, boundary, and fault-physics conditions are enforced while no conditions are enforced on the terminal boundary, $\partial\Omega_\mathrm{T}$. We further partition $\partial\Omega_\mathrm{B}$ as $\partial\Omega_\mathrm{B} = \overline{\partial\Omega_\mathrm{D} \cup \partial\Omega_\mathrm{N}}$ in which $\partial\Omega_\mathrm{D}$ and $\partial\Omega_\mathrm{N}$ are the parts of $\partial\Omega_\mathrm{B}$ on which Dirichlet and Neumann boundary conditions are enforced. Similarly, $\partial\Omega_\Gamma$ comprises both sides of the oriented fault manifolds in Γ, and we write $\partial\Omega_\Gamma = \partial\Omega_{\Gamma^+} \cup \partial\Omega_{\Gamma^-}$ to distinguish the positive and negative faces of the fault network.

The governing equations and initial-boundary conditions and their discrete treatment in the cSDG approximation are, for the most part, identical to the development in Chapter 2. The only exception is the jump conditions on the fault trajectory, $\partial\Omega_\Gamma$, which are developed below. Given that γ is a material interface, the fault trajectory Γ is restricted to a subset of spacetime orientations where the formulation of jump conditions does not require the machinery of differential forms and exterior calculus. Accordingly, we use direct tensor

[1] This convention agrees with the labeling used in [2], but is opposite to the labeling convention in [3]. This leads to a sign difference in the jump operator, $[\![\cdot]\!]$, introduced in 4.2.2 below. However, the formulations in the two papers are equivalent.

notation to formulate jump conditions across the fault trajectory below.

4.2.2 Jump Conditions Across Fault Trajectories

Discontinuities in solution fields may arise at physical interfaces, including faults, and across characteristic surfaces in the presence of weak shocks. They also arise in discontinuous Galerkin models due to weak enforcement of initial/boundary and inter-element continuity conditions. In these cases, the differential operators in the governing System (2.2) acquire jump parts that must be included in a complete representation of the system.

Miller *et al.* [20] use exterior calculus and differential forms to derive jump conditions for linear elastodynamics in $d \in \{1, 2, 3\}$ spatial dimensions across d-manifolds with arbitrary orientation in $\mathbb{E}^d \times \mathbb{R}$. Three classes of orientation-dependent jump conditions emerge: those involving linear combinations of $[\![\mathbf{S}]\!]$ and $[\![\mathbf{p}]\!]$, those involving linear combinations of $[\![\mathbf{E}]\!]$ and $[\![\mathbf{v}]\!]$, and those involving $[\![\mathbf{u}]\!]$ in which $[\![\cdot]\!]$ denotes the jump of the enclosed quantity across a given jump manifold.

The machinery of differential forms and exterior calculus is required to obtain objective formulations of the elastodynamic jump conditions across d-manifolds with arbitrary spacetime orientations. In particular, the lack of an inner-product on $\mathbb{E}^d \times \mathbb{R}$ precludes an objective definition of vectors normal to general spacetime d-manifolds. However, it is possible to formulate the cSDG seismic rupture model using standard tensor notation if we limit our attention to jump conditions across the fault trajectory Γ. We follow this approach from here on in this chapter to develop the cSDG rupture model. The results are compatible with the more general differential forms formulation presented in Section 2.4. We implemented the cSDG rupture model in a code based on that framework and used it to generate the numerical results presented in Section 4.4.

A fault trajectory, Γ, is comprised exclusively of *material-interface trajectories*; *i.e.,* d-manifolds on which (1) the purely spatial component of the differential, $d\Gamma$, vanishes, and (2) there exists a purely spatial normal vector, $\mathbf{n} \in \mathbb{E}^d$, consistent with the orientation of Γ — both almost everywhere on Γ. Applying these special properties to the three-field formulation in [20], we find that the jump condition components involving $[\![\mathbf{u}]\!]$, $[\![\mathbf{E}]\!]$, and $[\![\mathbf{p}]\!]$ are inactive on Γ, and the jump conditions simplify to

$$\left(\overset{\star}{\mathbf{S}} - \mathbf{S}^{\pm}\right)(\mathbf{n}) = \mathbf{0} \tag{4.1a}$$

$$\left(\overset{\star}{\mathbf{v}} - \mathbf{v}\right)^{\pm} = \mathbf{0} \tag{4.1b}$$

in which a superscript or subscript $\pm$ denotes a trace on the $+$ or $-$ side of Γ, and a superposed

$\star$ denotes a Riemann value on Γ. Thus, System (4.1) presents four distinct jump conditions relative to Riemann values, two for each side of the fault.

We assume zero external forces acting on Γ, so that the $+$ and $-$ sides of the fault share common Riemann values for the normal stress component, $\overset{\star}{\mathbf{S}}(\mathbf{n})$. Thus, we write $\overset{\star}{\mathbf{S}}$ in (4.1a) without further decoration. The velocity Riemann values on the $+$ and $-$ sides can be distinct due to possible non-vanishing slip and separation rates across the fault, so we retain the $\pm$ decoration on $\overset{\star}{\mathbf{v}}$ in (4.1b). In the single-field displacement formulation used in this work, we rewrite system (4.1) as modified jump conditions on Γ,

$$\left(\overset{\star}{\mathbf{S}} - [\mathbf{C}(\mathrm{sym}(\nabla\mathbf{u})]^{\pm}\right)(\mathbf{n}) = \mathbf{0} \tag{4.2a}$$

$$\left(\overset{\star}{\mathbf{v}} - \dot{\mathbf{u}}\right)^{\pm} = \mathbf{0} \tag{4.2b}$$

4.2.3 Dynamic Fault Response Model

The dynamic fault response on Γ is described in terms of a friction model with three *contact modes*: contact–stick, contact–slip, and separation. After introducing a local spatial coordinate frame on Γ in part 4.2.3, we formulate a slip-weakening friction model in 4.2.3 and summarize Riemann solutions by Abedi and Haber [2] for the complete set of contact modes in part 4.2.3. In contrast to quasi-static models, the dynamic jump conditions generated by the Riemann solutions in system (2.7) are consistent with the characteristic structure of elastodynamic hyperbolic systems. Abedi and Haber demonstrate that enforcing jump conditions relative to Riemann fluxes on both sides of a contact-interface trajectory improves accuracy and mitigates spurious numerical oscillations induced by impact events, sudden separation-to-contact transitions, and other sources of sharp wave fronts.

Local spatial coordinate frame and component representations

We establish a local Cartesian coordinate frame on Γ in $\mathbb{E}^d$ with local orthonormal vector and covector bases, $\mathbf{e}_i$ and $\mathbf{e}^i$ for $i \in \mathbb{N}^d := \{\mathbb{N}^+ : i \leq d\}$, such that $\mathbf{e}_1$ coincides with the local unit normal vector, $\mathbf{n}$, to Γ. The remaining basis vectors, $\mathbf{e}_i$ for $i \in \mathbb{N}^d \setminus 1$, span the local tangent space of Γ. We assign the labels, '$+$', to the fault-face trajectory whose outward normal, $\mathbf{n}^+$, coincides with $\mathbf{n}$ and '$-$' to the fault-face trajectory whose outward normal, $\mathbf{n}^-$, coincides with $-\mathbf{n}$.

We express kinematic quantities in component form with respect to the vector basis as,[2]

$$\mathbf{u} = u^i \mathbf{e}_i \tag{4.3a}$$

$$\mathbf{v} = v^i \mathbf{e}_i \tag{4.3b}$$

$$\mathbf{E} = E^{ij} \mathbf{e}_i \otimes \mathbf{e}_j \tag{4.3c}$$

and force-like quantities with respect to the covector basis as

$$\mathbf{p} = p_i \mathbf{e}^i \tag{4.4a}$$

$$\mathbf{S} = S_{ij} \mathbf{e}^i \otimes \mathbf{e}^j \tag{4.4b}$$

The component forms of the jump conditions on Γ in the local frame, system (4.1), reduce to

$$\left(\overset{\star}{S}_{1i} - S^{\pm}_{1i}\right) \mathbf{e}^i = \mathbf{0} \qquad \Leftrightarrow \qquad \overset{\star}{S}_{1i} - S^{\pm}_{1i} = 0 \tag{4.5a}$$

$$\left(\overset{\star}{v}{}^i_{\pm} - v^i_{\pm}\right) \mathbf{e}_i = \mathbf{0} \qquad \Leftrightarrow \qquad \overset{\star}{v}{}^i_{\pm} - v^i_{\pm} = 0 \tag{4.5b}$$

for $i \in \mathbb{N}^d$. Similar results hold for the three-field and one-field jump conditions, system (2.7).

Separation, slip, and slip-weakening friction model

We define a jump operator across Γ, $[\![\cdot]\!]$, such that $[\![f]\!] := f^- - f^+$ (or $f_- - f_+$). We decompose the displacement jump across Γ into normal and tangential parts,

$$[\![\mathbf{u}]\!] = \hat{\delta}\mathbf{e}_1 + \delta \mathbf{e}_\delta \tag{4.6}$$

in which $\hat{\delta}$, δ, and $\mathbf{e}_\delta$ are, respectively, the normal separation across Γ, the slip magnitude, and the unit vector in the direction of slip given by

$$\hat{\delta} := [\![u^1]\!] \tag{4.7a}$$

$$\delta := \left| \sum_{i \in \mathbb{N}^d \setminus 1} [\![u^i]\!] \mathbf{e}_i \right| \tag{4.7b}$$

$$\mathbf{e}_\delta := \frac{1}{\delta} \sum_{i \in \mathbb{N}^d \setminus 1} [\![u^i]\!] \mathbf{e}_i \tag{4.7c}$$

[2] Abedi, Haber, & Petracovici [1] and Miller *et al.* [20] defined kinematic and force-like quantities with respect to the covector and vector bases, respectively. We reverse that convention from here on to better conform with common practice.

The slip-rate magnitude and direction are given by

$$\dot{\delta} := \left| \sum_{i \in \mathbb{N}^d \setminus 1} [\![v^i]\!] \mathbf{e}_i \right| \tag{4.8a}$$

$$\mathbf{e}^v := \frac{1}{\dot{\delta}} \sum_{i \in \mathbb{N}^d \setminus 1} \mathbf{g}\left([\![v^j]\!]\mathbf{e}_j\right) \Leftrightarrow e_i^{\mathbf{v}} = \begin{cases} 0 & i = 1 \\ \delta_{ij} \frac{[\![v^j]\!]}{\dot{\delta}} & i \in \mathbb{N}^d \setminus 1 \end{cases} \tag{4.8b}$$

We decompose the Riemann surface traction vector, $\overset{\star}{s} := \overset{\star}{S}_{1i}\mathbf{e}^i$, into normal and tangential components

$$\overset{\star}{\mathbf{s}}^{\pm} = \pm \left[-\sigma \mathbf{e}^1 + \tau \mathbf{e}^\tau \right] \tag{4.9}$$

in which σ, τ, and $\mathbf{e}^\tau$ are, respectively, the normal compression component of surface traction vector, the magnitude of the friction vector, and the unit vector in the direction of the friction vector.

$$\sigma := -\overset{\star}{S}_{11} \tag{4.10a}$$

$$\tau := \left| \sum_{i \in \mathbb{N}^d \setminus 1} \overset{\star}{S}_{1i} \mathbf{e}^i \right| \tag{4.10b}$$

$$\mathbf{e}^\tau := \frac{1}{\tau} \sum_{i \in \mathbb{N}^d \setminus 1} \overset{\star}{S}_{1i} \mathbf{e}^i \Leftrightarrow e_i^\tau = \begin{cases} 0 & i = 1 \\ \frac{\overset{\star}{S}_{1i}}{\tau} & i \in \mathbb{N}^d \setminus 1 \end{cases} \tag{4.10c}$$

From here on we adopt an isotropic, bilinear slip-weakening friction model [79],

$$\tau \leq k(\delta) \left\langle \sigma \right\rangle_+ \tag{4.11a}$$

$$k(\delta) := \begin{cases} k_\mathrm{s} - \frac{\delta}{\delta_\mathrm{c}}(k_\mathrm{s} - k_\mathrm{d}) & 0 \leq \delta < \delta_\mathrm{c} \\ k_\mathrm{d} & \delta \geq \delta_\mathrm{c} \end{cases} \tag{4.11b}$$

in which k_s and k_d are the static and dynamic coefficients of friction, δ_c is the slip length scale, and $\langle \cdot \rangle_+$ denotes the positive Macaulay bracket. Although somewhat rudimentary, bilinear slip-weakening models are commonly used to model rupture in geological materials, as in the benchmark verification problems in Section 4.4.

Riemann solutions for dynamic contact modes on fault trajectories

This section summarizes Riemann solutions for $\overset{\star}{S}_{1i}$ and $\left(\overset{\star}{v}^{i}\right)^{\pm}$ for each dynamic contact mode, as derived in [2, 3]. For simplicity, we assume isotropic elastic response, but allow for distinct bulk material properties — specifically Lamé parameters, $\lambda^{\pm}$ and $\mu^{\pm}$, and mass density, $\rho^{\pm}$ — on opposite sides of the fault. We obtain local impedance components,

$$Z_i^{\pm} = \begin{cases} (c_{\mathrm{d}}\rho)^{\pm} & i = 1 \\ (c_{\mathrm{s}}\rho)^{\pm} & i \in \mathbb{N}^d \setminus 1 \end{cases} \tag{4.12}$$

in which the local dilatational and shear wave speeds are, respectively,

$$c_{\mathrm{d}}^{\pm} = \sqrt{\frac{\lambda^{\pm} + 2\mu^{\pm}}{\rho^{\pm}}}, \quad c_{\mathrm{s}}^{\pm} = \sqrt{\frac{\mu^{\pm}}{\rho^{\pm}}} \tag{4.13}$$

Contact–stick mode: The Riemann solutions for this mode are identical to those for a perfectly bonded interface and require zero separation ($\hat{\delta} = 0$) and velocity continuity across Γ ($[\![\overset{\star}{\mathbf{v}}]\!] = \mathbf{0} \Leftrightarrow \overset{\star}{\mathbf{v}}^{+} = \overset{\star}{\mathbf{v}}^{-} =: \overset{\star}{\mathbf{v}}$). Moreover, the Riemann solution,

$$\overset{\star}{S}_{1i} = \frac{S_{1i}^{+}Z_i^{-} + S_{1i}^{-}Z_i^{+}}{Z_i^{-} + Z_i^{+}} + \frac{Z_i^{-}Z_i^{+}}{Z_i^{-} + Z_i^{+}}[\![v^i]\!] =: \hat{S}_{1i} \tag{4.14a}$$

$$\overset{\star}{v}^{i} = \frac{S_{1i}^{-} - S_{1i}^{+}}{Z_i^{-} + Z_i^{+}} + \frac{v_{+}^{i}Z_i^{+} + v_{-}^{i}Z_i^{-}}{Z_i^{-} + Z_i^{+}} =: \hat{v}^i \tag{4.14b}$$

in which $i \in \mathbb{N}^d$ and there are no sums on i, must satisfy $\sigma \geq 0$ and the friction inequality, eq. (4.11a).

Contact–slip mode: This mode requires zero separation ($\hat{\delta} = 0 \Rightarrow [\![v^1]\!] = 0$), so the normal components of the contact–slip Riemann solution, $\overset{\star}{S}_{11}$ and $\overset{\star}{v}^{1}$, are the same as in contact–stick mode. The tangential velocity components may be discontinuous across the fault to accommodate non-zero slip rates, while the tangential Riemann stress components must satisfy eq. (4.11a) by equality. The Riemann solution, which must also satisfy $\sigma \geq 0$ to maintain contact, is given by

$$\overset{\star}{S}_{1i} = \begin{cases} \hat{S}_{11} & i = 1 \\ k(\delta)\, \langle\sigma\rangle_+\, e^\tau_i & i \in \mathbb{N}^d \setminus 1 \end{cases} \tag{4.15a}$$

$$\overset{\star}{v}{}^i_\pm = \begin{cases} \hat{v}^1 & i = 1 \\ v^i_\pm \pm \delta^{ij} \frac{\overset{\star}{S}_{1j} - S^\pm_{1j}}{Z^\pm_i} & i \in \mathbb{N}^d \setminus 1 \end{cases} \tag{4.15b}$$

in which $\hat{S}_{11}$ and $\hat{v}^1$ are defined in (4.14), $\overset{\star}{S}_{1j}$ in (4.15b) is defined in the second branch of (4.15a), and there are no sums on i.

System 4.15 follows a novel formulation of the contact–slip Riemann solution introduced in [2] In combination with the jump conditions (4.1), this formulation offers several advantages relative to previous formulations that significantly simplify its numerical implementation:

1. It enforces equal normal velocities on both fault faces to preserve contact without resorting to Lagrange multiplier or other constraint-enforcement techniques.
2. It avoids nonlinear coupling between the tangential parts of the Riemann solutions for stress and velocity that requires iterative solution strategies in previous formulations.
3. It substitutes the continuous direction covector, $\mathbf{e}^\tau$, for the discontinuous covector, $\mathbf{e}^v$, in the frictional part of the Riemann stress solution in (4.15a).[3]

Separation mode: Velocity is fully discontinuous across the fault trajectory in this mode, and the normal part of the Riemann stress is determined by prescribed, self-equilibrating traction vectors, $\bar{\mathbf{s}}^\pm$, that act on the fault-face trajectories such that $\bar{\mathbf{s}} := \bar{\mathbf{s}}^+ = -\bar{\mathbf{s}}^-$. The Riemann solution for separation mode is:

$$\overset{\star}{S}_{1i} = \bar{s}_i \tag{4.16a}$$

$$\overset{\star}{v}{}^i_\pm = v^i_\pm \pm \frac{\bar{s}_i - S^\pm_{1i}}{Z^\pm_i} \tag{4.16b}$$

in which there are no sums on $i \in \mathbb{N}^d$.

[3]The covector $\mathbf{e}^v$ is discontinuous at contact–stick to contact–slip transitions where $[\![\mathbf{v}]\!] = \mathbf{0}$, and this significantly complicates previous numerical implementations. However, this simplifying substitution is only valid for isotropic friction models.

4.3 The cSDG Rupture Model

4.3.1 Formulation

We restrict our attention to numerical aspects of the dynamic rupture model within the cSDG framework. The interested reader can find coverage of other aspects of the numerical formulations for the 1-field and 3-field cSDG models for linear elastodynamics with contact in [1, 20, 2].

We decompose the overall solution into static and dynamic parts. The time-invariant static part corresponds to an initial equilibrium stress field acting on the system at rest with vanishing deformation, while the dynamic part includes the complete kinematic response. Following common practice, we prescribe a system of self-equilibrating static tractions acting on the fault network leaving the corresponding static stress field undefined. The static tractions, in combination with the slip-weakening friction model, are sufficient to drive the dynamic response which we compute with the adaptive cSDG method. We compute total fault tractions for use in the slip-weakening model by superposing the prescribed static and computed dynamic tractions; *cf.* eq. (4.11a).

We introduce a smooth regularization of the bilinear slip-weakening model, eq. (4.11b), using Hermite interpolations and a regularization parameter, $\epsilon \in [0, 0.5]$, to facilitate convergence of Newton-Raphson iterations within our numerical procedure. Defining regularization zones with half-widths, $\epsilon\delta_c$, centered at $\delta = 0$ and $\delta = \delta_c$ while enforcing zero slope at $\delta = 0$ and $\delta = \delta_3$, we obtain

$$k(\delta) = \begin{cases} \breve{k}(\delta) & 0 < \delta < \delta_1 \\ k_s - \frac{\delta}{\delta_c}(k_s - k_d) & \delta_1 \le \delta < \delta_2 \\ \tilde{k}(\delta) & \delta_2 \le \delta < \delta_3 \\ k_d & \delta \ge \delta_3 \end{cases} \tag{4.17a}$$

$$\begin{aligned} \breve{k}(\delta) := {} & k_s(2\breve{\delta}^3 - 3\breve{\delta}^2 + 1) + (k_s + m\delta_1)(-2\breve{\delta}^3 + 3\breve{\delta}^2) \\ & + m\delta_1(\breve{\delta}^3 - \breve{\delta}^2) \end{aligned} \tag{4.17b}$$

$$\begin{aligned} \tilde{k}(\delta) := {} & (k_s + m\delta_2)(2\tilde{\delta}^3 - 3\tilde{\delta}^2 + 1) \\ & + m(\delta_3 - \delta_2)(\tilde{\delta}^3 - 2\tilde{\delta}^2 + \tilde{\delta}) + k_d(-2\tilde{\delta}^3 + 3\tilde{\delta}^2) \end{aligned} \tag{4.17c}$$

in which $\delta_1 := \epsilon\delta_c$, $\delta_2 := (1-\epsilon)\delta_c$, $\delta_3 := (1+\epsilon)\delta_c$, $m := (k_d - k_s)/\delta_c$, and we define normalized slip functions, $\breve{\delta}(\delta) := \delta/\delta_1$ and $\tilde{\delta}(\delta) := (\delta - \delta_2)/(\delta_3 - \delta_2)$.[4] Figure 4.2 illustrates convergence

[4]We suppress the dependence of $\breve{\delta}$ and $\tilde{\delta}$ on δ to simplify the expressions in (4.17b) and (4.17c).

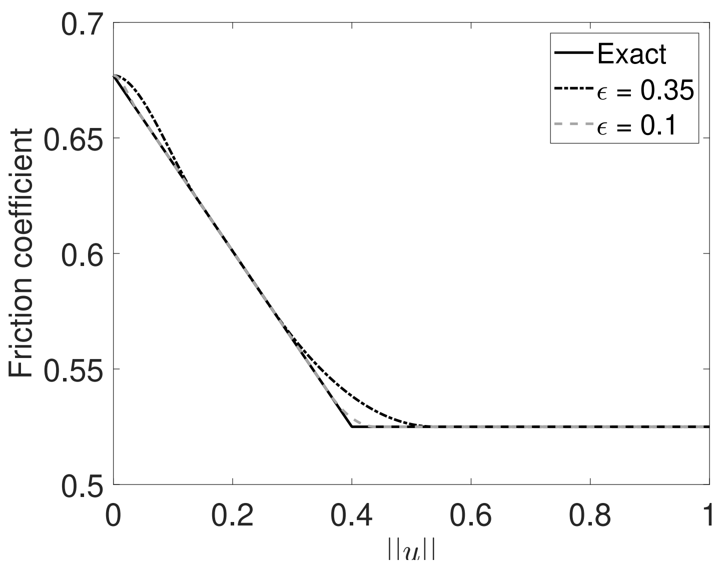


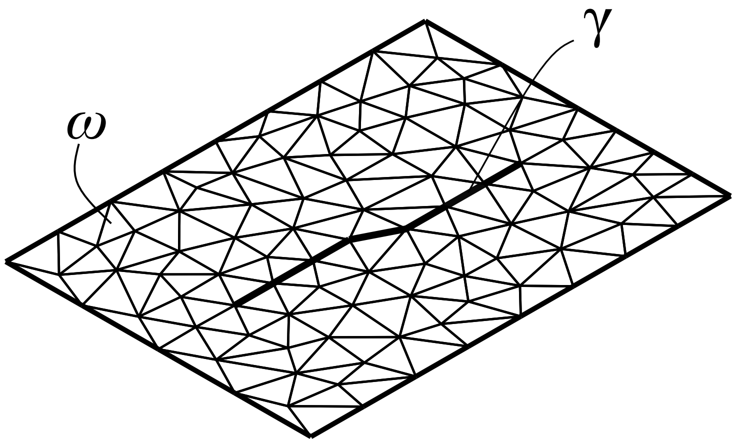
Figure 4.2: Convergence of regularized bilinear slip-weakening model as the regularization parameter, ϵ, tends to 0 for $k_{\text{s}} = 0.677$, $k_{\text{d}} = 0.525$, and $\delta_{\text{c}} = 0.4$.

of the regularized slip-weakening model to eq. (4.11b) as $\epsilon \to 0$. We use a smaller value of the regularization parameter, $\epsilon = 0.025$, in the numerical examples in Section 4.4 for which the difference between the exact and regularized slip-weakening relations would be barely visible in the figure.

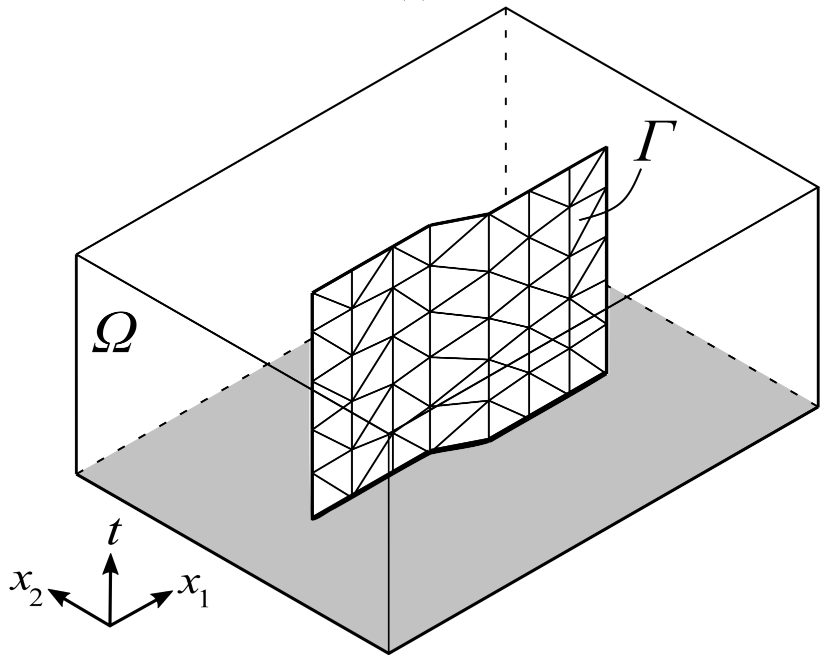
4.3.2 Implementation on Causal Spacetime Meshes

This subsection describes the cSDG method's advancing-front spacetime meshing procedure with specializations for modeling earthquake ruptures in greater detail. We seek an unstructured discretization of Ω with a set of spacetime simplex elements, $\{Q_\alpha\}$, grouped into *patches*. The facets of each patch must be causal, as defined is Section 2.2.1. This causality constraint ensures that information travels in only one direction across each patch facet, either inflow or outflow, and implies an asymmetric dependency graph between patch-wise solutions consistent with the causality principle of physics and the characteristic structure of hyperbolic systems. A spacetime mesh whose patches satisfy the causality constraint is called a *causal mesh*.

In combination with spacetime discontinuous Galerkin discretizations of the solution fields, causal meshes generate partial orderings of patches through which a locally implicit, unconditionally stable problem can be constructed on each patch whose solution depends only on prescribed initial/boundary data and solutions on adjacent previously-solved patches; *cf.* [1, 20]. This structure leads to linear complexity in the number of spacetime patches.

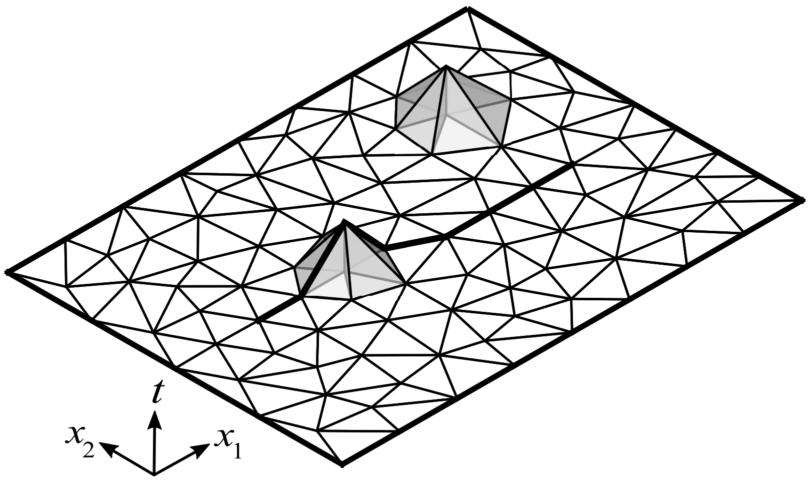


(a)

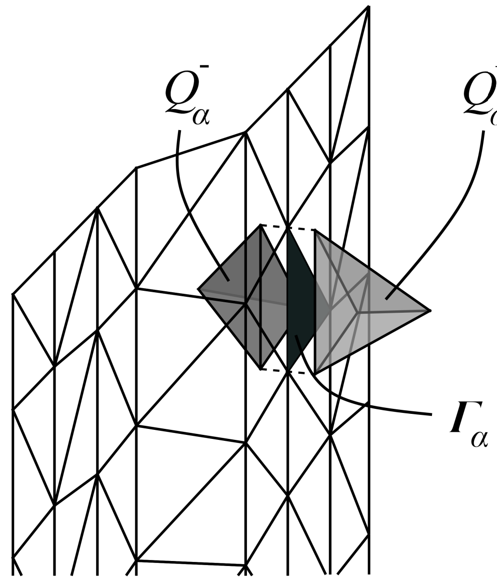


(b)

Figure 4.3: (a) Triangulation of initial front at time $t = 0$ and, (b), triangulation of fault trajectory, Γ, induced by tetrahedral mesh on Ω.



(a)



(b)

Figure 4.4: Early-stage tent pitching and triangulation of fault trajectory: (a) Early front configuration after two tent-pitching operations create patches that contain tetrahedral spacetime elements and, in one case, a pair of triangular spacetime cells on the fault trajectory; (b) Facets of adjacent tetrahedral spacetime elements, $Q_\alpha^\pm$, generate a triangular spacetime cell, Γ_α, on the fault trajectory as part of a conforming triangulation of Γ.

Moreover, discrete cSDG solutions balance linear and angular momentum over each spacetime element to within machine precision.

We use the adaptive Tent Pitcher method to construct causal spacetime meshes by advancing through Ω a space-like *front mesh* in which each vertex carries an independent time coordinate, as shown in Fig. 4.3a for a two-dimensional problem. Given an initial front mesh, *i.e.,* a spatial triangulation of ω at $t = 0$, Tent Pitcher constructs a spacetime mesh on Ω, one patch at a time. The basic tent pitching operation advances a vertex in time subject to the causality constraint to define a new front configuration. We mesh the spacetime region between the previous and new fronts with simplex elements to form a new patch, immediately solve the localized patch problem[5], and update the current front to include the patch's outflow facets in its new configuration. This process repeats until every front vertex passes the final time of the simulation and patch-wise solutions cover Ω. Figure 4.4a depicts the updated front configuration and two patches comprised of tetrahedral spacetime elements after two tent-pitching operations.

In contrast to conventional time-marching schemes in which the solution advances synchronously via globally-uniform time steps, cSDG solutions advance locally and asynchronously at a much finer granularity. While synchronous time-step sizes are limited by global worst-case accuracy or stability constraints, the durations of individual cSDG patches are limited by local causality constraints and local accuracy considerations.[6]

The Tent Pitcher meshing process generates a simplicial mesh on Γ that conforms with the spacetime mesh on Ω. That is, each simplex on Γ coincides with facets of adjacent simplex elements, Q_α^+ and Q_α^-, on Ω, as shown in Fig. 4.4b. Thus, patches that straddle the fault trajectory cover Γ with simplex cells as the solution progresses. This provides a convenient computational framework for coupling fault rupture models with the bulk dynamic response.

4.3.3 Adaptive Spacetime Meshing

A new approach to dynamic adaptive meshing

Adaptive remeshing operations in synchronous schemes are expensive global operations that typically execute once every $N \geq 1$ time steps. In contrast, cSDG solvers support inexpensive local adaptive operations that execute at the same patch-level granularity as the localized spacetime-meshing and patch-solution operations. Thus, the number of potential adaptive operations per layer of patches is roughly equal to the number of vertices in the current front.

[5]Inter-element facets within a patch are generally not space-like, so the element-wise solutions within each patch are coupled and must be solved simultaneously as a patch-wise localized problem.

[6]Localized patch problems are implicit and unconditionally stable, so patch durations are not limited by stability constraints.

These properties equip cSDG solvers with a highly dynamic and efficient form of adaptive spacetime meshing capable of capturing fast-moving wave fronts and rupture tips, spanning wide ranges of spatio-temporal scales in multi-scale simulations, and continually improving mesh quality as a solution progresses.

Dynamic adaptive remeshing for transient problems is typically implemented between synchronous time steps and requires a projection of the solution from the old mesh onto the new mesh — an operation that generally limits transient solutions to first-order accuracy. The cSDG solver implements adaptive edge-flip, coarsening, and mesh-smoothing operations as special spacetime patches whose inflow and outflow faces conform, respectively, to the old and new front meshes. Adaptive cSDG mesh refinement is a local operation that involves discarding patch solutions that fail their error-indicator checks, local refinement of the previous front mesh, and restarting the spacetime meshing/solution process. No solution projections are required, so cSDG adaptive meshing procedures preserve the full order of accuracy of high-order discontinuous Galerkin models. Detailed descriptions of cSDG adaptive meshing procedures appear in [18] and [15], including an adaptive error indicator that controls numerical dissipation error in the bulk elastodynamic response.

Adaptive error indicators for the fault rupture model

We augment the bulk dissipation error indicator with specialized error indicators for controlling the accuracy of the cSDG rupture response model. These indicators measure discrepancies between errors in separation rate, slip rate, normal stress, and shear stress between Riemann values and traces of the bulk solution from both sides of the fault trajectory, Γ. We define the following scalar jump operators for velocity

$$[\![\mathbf{v}^{\mathrm{n}}]\!]^{\star}_{\pm} := \left|\left(\overset{\star}{v}{}^{1} - v^{1}_{\pm}\right)\mathbf{e}_1\right| \tag{4.18a}$$

$$[\![\mathbf{v}^{\mathrm{t}}]\!]^{\star}_{\pm} := \left|\sum_{i\in\mathbb{N}^d\setminus 1}\left(\overset{\star}{v}{}^{i} - v^{i}_{\pm}\right)\mathbf{e}_i\right| \tag{4.18b}$$

and require $\forall\,\Gamma_\alpha \in \Gamma$ that

$$\frac{\left\|[\![\mathbf{v}^{\mathrm{n}}]\!]^{\star}_{\pm}\right\|_{l^\infty(\Gamma_\alpha)}\Delta t_\alpha}{\bar{d}} < \kappa^{\mathrm{n}}_{\mathbf{v}} \ll 1 \tag{4.19a}$$

$$\frac{\left\|[\![\mathbf{v}^{\mathrm{t}}]\!]^{\star}_{\pm}\right\|_{l^\infty(\Gamma_\alpha)}\Delta t_\alpha}{\delta_c} < \kappa^{\mathrm{t}}_{\mathbf{v}} \ll 1 \tag{4.19b}$$

in which the $l^\infty(\Gamma_\alpha)$-norm is taken over all quadrature points on Γ_α that are either in pure contact mode or transitioning from separation to contact, Δt_α is the duration of Γ_α, $\bar{d}$ is the upper limit of the separation regularization zone in which the lower limit is $\underline{\mathrm{d}} = 0$, δ_c is the slip-weakening distance, while $\kappa_{\mathbf{v}}^{\mathrm{n}}$ and $\kappa_{\mathbf{v}}^{\mathrm{t}}$ are user-selected error tolerances for controlling velocity-related mismatch between Riemann values and traces of the bulk cSDG solution on the fault.

We apply a similar approach to the normal and tangential components of the surface stress error. That is, we define

$$[\![\mathbf{S}_{\mathrm{n}}]\!]_{\star}^{\pm} := \left| \left(\overset{\star}{S}_{11} - S_{11}^{\pm} \right) \mathbf{e}^1 \right| \tag{4.20a}$$

$$[\![\mathbf{S}_{\mathrm{t}}]\!]_{\star}^{\pm} := \left| \sum_{i \in \mathbb{N}^d \setminus 1} \left(\overset{\star}{S}_{1i} - S_{1i}^{\pm} \right) \mathbf{e}^i \right| \tag{4.20b}$$

and require $\forall\, \Gamma_\alpha \in \Gamma$ that

$$\frac{\| [\![\mathbf{S}_{\mathrm{n}}]\!]_{\star}^{\pm} \|_{l^\infty(\Gamma_\alpha)}}{\bar{\sigma}} < \kappa_{\mathrm{n}}^{\mathbf{S}} \ll 1 \tag{4.21a}$$

$$\frac{\| [\![\mathbf{S}_{\mathrm{t}}]\!]_{\star}^{\pm} \|_{l^\infty(\Gamma_\alpha)}}{\bar{\tau}} < \kappa_{\mathrm{t}}^{\mathbf{S}} \ll 1 \tag{4.21b}$$

in which $\bar{\sigma}$ is the initial compression acting across the fault, $\bar{\tau}$ is the shear strength associated with $\bar{\sigma}$, while $\kappa_{\mathrm{n}}^{\mathbf{S}}$ and $\kappa_{\mathrm{t}}^{\mathbf{S}}$ are user-selected error tolerances for controlling stress-related mismatch at the fault.

4.4 Numerical Verification, High-Resolution Capabilities, and Contact–Separation Model

In this section, we verify the cSDG dynamic rupture/contact model and demonstrate its high-order modeling and multi-scale adaptive meshing capabilities with a pair of two-dimensional Southern California Earthquake Center (SCEC) benchmark problems [80]. We compare cSDG results to solutions obtained with selected state-of-the-art methods, as posted in the SCEC repository or published in the recent literature. In contrast to most other methods, Miller *et al.* [20] showed that cSDG models gain computational efficiency — in addition to higher-order accuracy — with increasing polynomial order of the spacetime basis functions. We use the 1-field formulation with twelfth-order spacetime polynomial basis functions to generate the cSDG solutions in all numerical examples in this subsection.

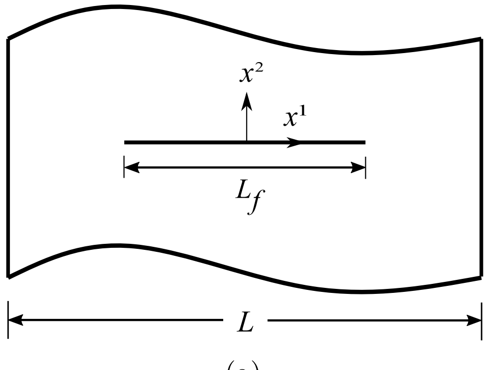
4.4.1 SCEC TPV205-2D Benchmark

Problem description

This problem involves a vertical strike-slip fault of length L_f centered in a domain of width L comprised of homogeneous rock modelled as an isotropic, linear-elastic solid under plane strain conditions, as depicted in Fig. 4.5a. Wave speeds, c_d and c_s, and mass density, ρ, determine the rock's mechanical properties. The regularized bilinear slip-weakening model of Section 4.3.1 — with slip-weakening distance, δ_c, and static and dynamic coefficients of friction, k_s and k_d — determine the fault's frictional response. The rock is assumed to have sufficient strength to preclude rupture beyond the initial fault configuration.

Uniform static normal compressive tractions of magnitude σ^0 act across the fault for its entire length. Due to the symmetry of the analysis domain and the anti-symmetry of the shear tractions about the x^1-axis, the dynamic normal tractions vanish everywhere on the fault at all times. Therefore, the total normal tractions, $\sigma = \sigma^0$, are invariant in space and time, as are the static and dynamic shear resistances: $\overline{\tau} = k_\mathrm{s}\sigma^0$ and $\tilde{\tau} = k_\mathrm{d}\sigma^0$. The piece-wise constant distribution of right-lateral, static shear tractions, τ^0, depicted in Figure 4.5b drives the dynamic rupture process. A background static shear intensity, τ^0_b, holds over most of the fault with the exception of three 3km-wide zones. The left and right zones, centered at $x^1 = -7.5$km and $x^1 = 7.5$km, carry static shear tractions, τ^0_l and τ^0_r. A nucleation zone centered at $x^1 = 0$km carries static shear tractions, τ^0_n, that slightly exceed $\overline{\tau}$. This condition initiates instantaneous rupture across the entire nucleation zone at time $t = 0$. Numerical values for all model parameters appear in Table 4.1.

Figure 4.6a depicts the initial front mesh generated for the TPV205-2D problem. We specify a coarse initial front mesh comprised of 38 vertices and 66 elements in which seven edges, *cf.* Fig. 4.6b, represent the seven segments of piecewise-constant static shear traction values, τ^0, along the fault. We depend on the cSDG method's powerful dynamic adaptive meshing capabilities to automatically provide sufficient mesh refinement to capture traveling wavefronts and the fine details of rupture nucleation and propagation — down to the smallest relevant spatial and temporal scales, including those associated with rupture-process zones and even solution features generated by nonlinear response within process zones (*cf.* part 4.4.2 below). This relieves the user of the substantial burden of manually generating graded meshes and boosts computational efficiency well above what can be attained in seismic codes with non-adaptive graded meshes or, in very few cases, traditional adaptive meshing techniques.



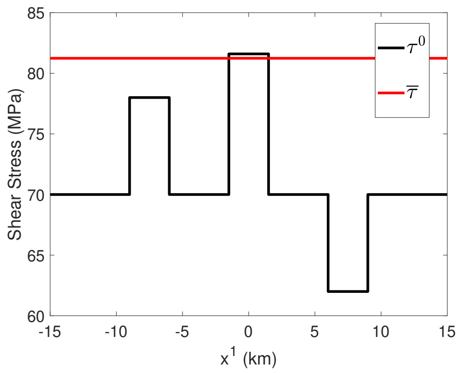


Figure 4.5: TPV205-2D problem description: (a) Schematic of spatial domain with central fault; (b) Distributions of static shear traction, τ^0, and time-invariant static shear strength, $\overline{\tau}$, along the fault.

Table 4.1: TPV205-2D model parameters

Parameter	Symbol	Value
Domain width (km)	L	100
Fault length (km)	L_f	30
Dilatational wave speed (m/s)	c_d	6000
Shear wave speed (m/s)	c_s	3464
Mass density (kg/m^3)	ρ	2670
Slip-weakening distance (m)	δ_c	0.4
Static coefficient of friction	k_s	0.677
Dynamic coefficient of friction	k_d	0.525
Slip-weakening regularization parameter	ϵ	0.025
Uniform static compressive traction (MPa)	σ^0	120
Background static shear traction (MPa)	τ^0_b	70.0
Static shear traction (left zone) (MPa)	τ^0_l	78.0
Static shear traction (right zone) (MPa)	τ^0_r	62.0
Static shear traction (nucleation zone) (MPa)	τ^0_n	81.6
Invariant static shear strength (MPa)	$\overline{\tau}$	81.24
Invariant dynamic shear resistance (MPa)	$\tilde{\tau}$	63.0
Normal velocity error tolerance	$\kappa^\mathrm{n}_\mathbf{v}$	1×10^{-6}
Tangential velocity error tolerance	$\kappa^\mathrm{t}_\mathbf{v}$	1×10^{-6}
Normal stress error tolerance	$\kappa^\mathbf{S}_\mathrm{n}$	1×10^{-8}
Tangential stress error tolerance	$\kappa^\mathbf{S}_\mathrm{t}$	1×10^{-8}

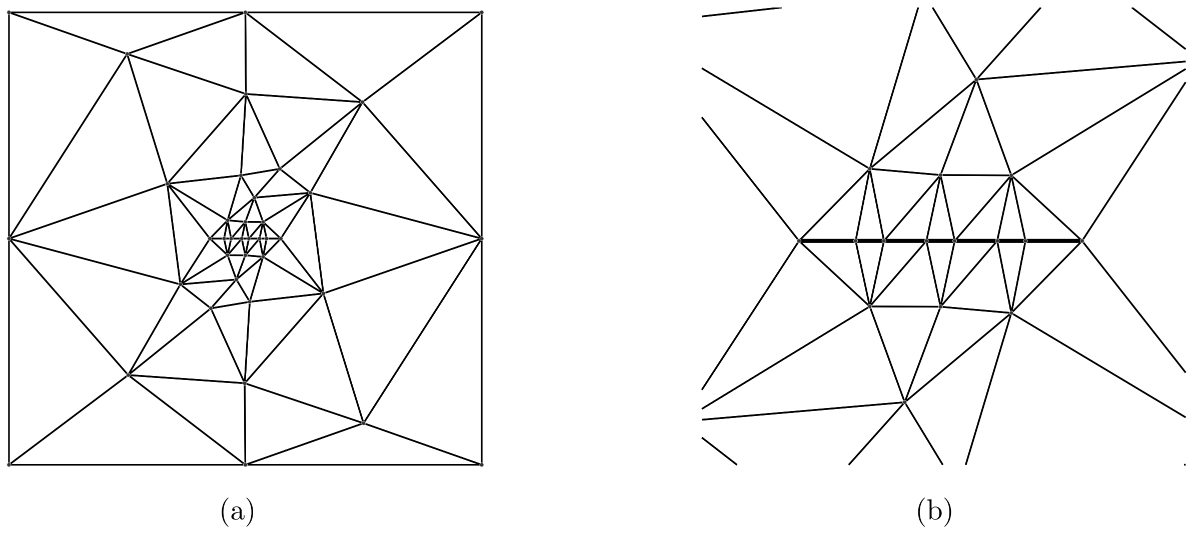
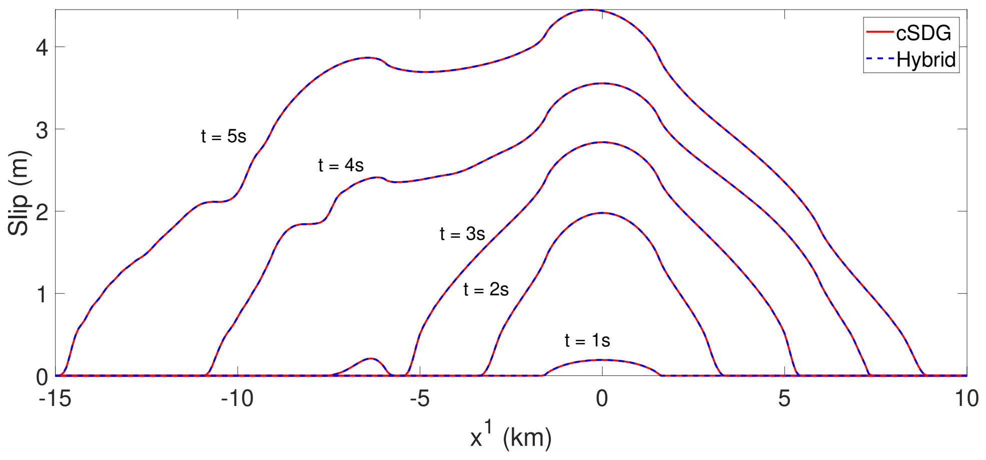


Figure 4.6: Initial front mesh for TPV205-2D at $t = 0$. (a) Overall mesh covering a 100km $\times$ 100km spatial domain with 38 vertices and 66 elements; (b) Detail of fault-region mesh in which the seven highlighted edges correspond to the seven segments of piecewise-constant static shear traction, τ^0, along the fault.

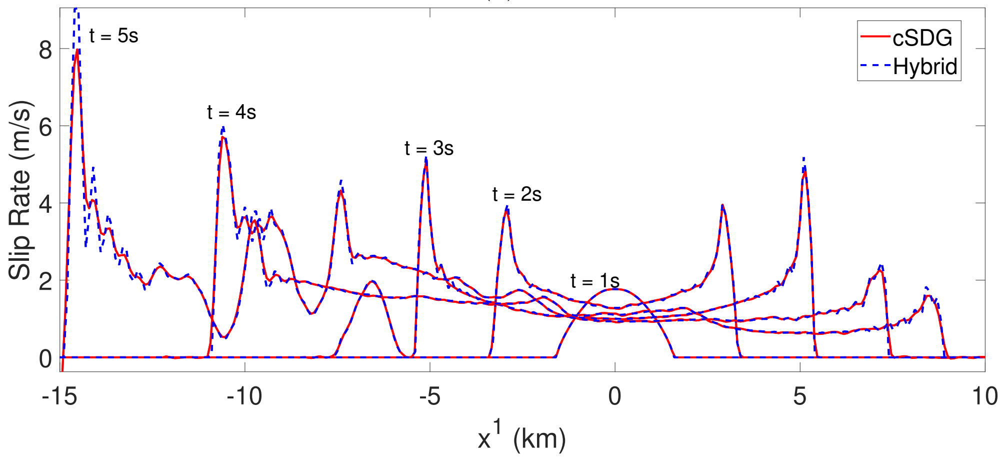
Verification of cSDG seismic rupture model

We verify the proposed rupture model by comparing cSDG results to solutions posted in the SCEC Benchmark Database [80] obtained with the hybrid finite element and spectral boundary integral (SBI) model of Ma *et al.* [69] and the high-order-accurate finite difference model, FDMAP, by Kozdon, Dunham, and Nordström [81]. From here on, we refer to these models, respectively, as the *Hybrid* and *FDMAP* models. The Hybrid model couples a high-resolution finite element discretization of the fault region with an SBI discretization at the domain boundary. A consistent exchange of displacement and boundary-traction components defines the coupling between the models. The FDMAP method uses summation-by-parts finite difference operators and coordinate mappings of multi-block mehes to model large fault systems with irregular geometries. Coupling conditions between mesh blocks that support bi-material interfaces provides FDMAP with a robust model for material heterogeneity.

Figure 4.7 compares snapshots of slip and slip rate distributions along the fault for the cSDG and Hybrid models. Figure 4.8 compares histories of slip and slip rate at selected station points along the fault located at ±4.5km and ±7.5km from the fault's center. The cSDG model accurately captures rupture nucleation, propagation and arrest. There are no visible differences between the cSDG and Hybrid results for slip. Although there is generally good agreement between the slip-rate solutions, we do observe some local differences. We attribute these to spurious oscillations and over/under-shoot in the Hybrid solution, as explained below.



(a)



(b)

Figure 4.7: Slip and slip-rate distributions for TPV205-2D along the fault at 1s intervals for cSDG and Hybrid models.

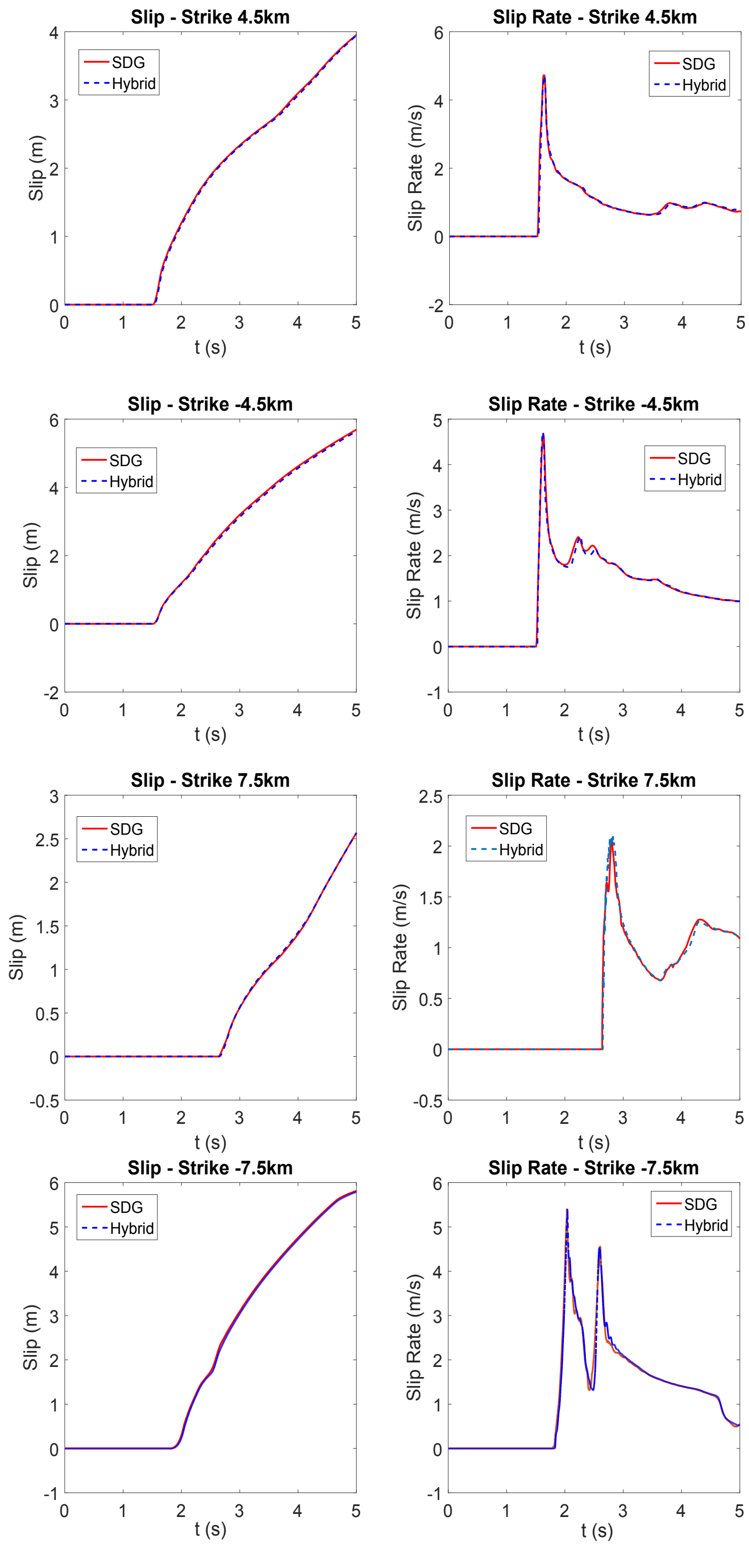


Figure 4.8: Slip and slip-rate histories for TPV205-2D at $x^1 = \pm 4.5$km and $x^1 = \pm 7.5$km by cSDG and Hybrid models.

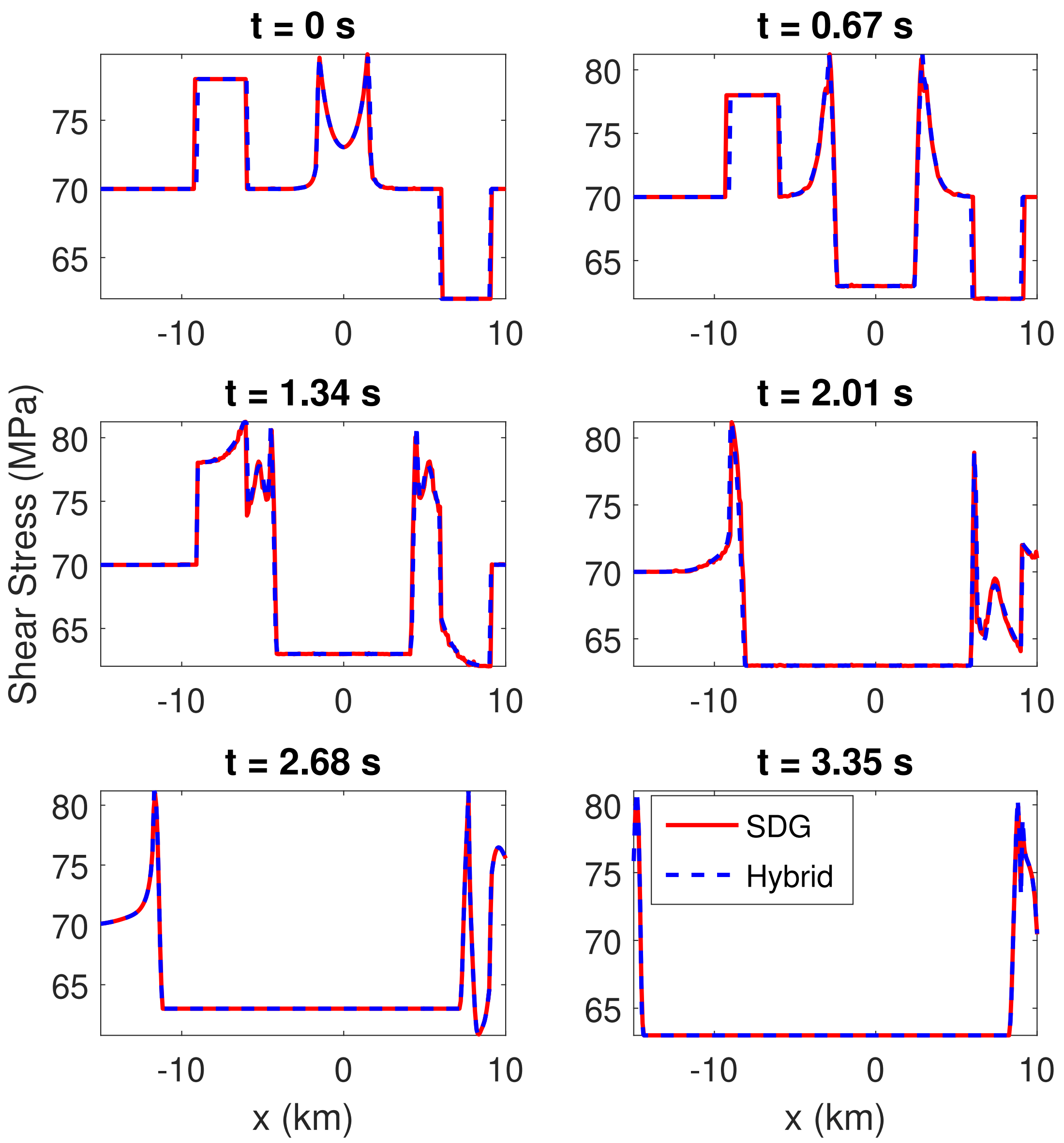


Figure 4.9: Shear Stress distributions for TPV205-2D along the fault for cSDG and Hybrid models.

High-resolution modeling capabilities

Under-resolved rupture simulations can mask high-frequency features and frustrate reliable evaluation of critical quantities of interest, such as slip rate and material acceleration. Unfortunately, factors such as overly coarse spatiotemporal discretization and filtration of spurious numerical oscillations commonly limit resolution in dynamic rupture simulations. Since higher resolution models generally incur greater computational expense, achieving high computational efficiency is an especially important concern in rupture modeling. We next compare the high-resolution performance of the Hybrid, FDMAP, and cSDG methods, each of which follows distinct strategies for boosting computational efficiency.

The Hybrid method couples a spectral boundary integral model of far-field response with a low-order, conforming finite element model that captures near-field response in a small region containing the faults. A consistent exchange of displacement and boundary-traction components defines the coupling between the models. The limited extent of the refined finite element mesh reduces computational expense. However, even with these savings, cost considerations still limit the near-field model's resolution, and Rayleigh damping is required to control spurious grid-scale oscillations.

FDMAP models combine arbitrarily high-order summation-by-parts (SBP) finite difference operators on multi-block meshes with fourth-order Runge-Kutta integrators in time to achieve up to fourth-order accuracy in a discrete SBP energy norm. Block-wise coordinate mappings and inter-block coupling conditions with support for bi-material interfaces provide robust modeling of material heterogeneity and fault networks with irregular geometry. However, overly strong coordinate transformations can compromise the stability of conservative SBP operators[7], so FDMAP uses non-conservative operators to ensure energy stability. The FDMAP coupling conditions preserve characteristic structure between blocks for bonded interfaces as well as contact–slip and contact–stick conditions across faults, so all fault contact modes are covered except separation. Although structure-preserving conditions are not enforced within blocks, the method's high-order accuracy contributes to its high-resolution modeling capabilities.

Multiple features of cSDG models combine to support high-resolution, multi-scale seismic simulations. For example, discontinuous Galerkin basis functions combined with causal spacetime meshing localize the asynchronous solution process to a partial ordering of implicit, unconditionally-stable, patch-wise problems with linear computational complexity in the number of patches and machine-precision conservation of linear and angular momentum over every spacetime element. The cSDG jump conditions preserve characteristic structure across

[7]See [81] for the case of models with uniform elastic coefficients.

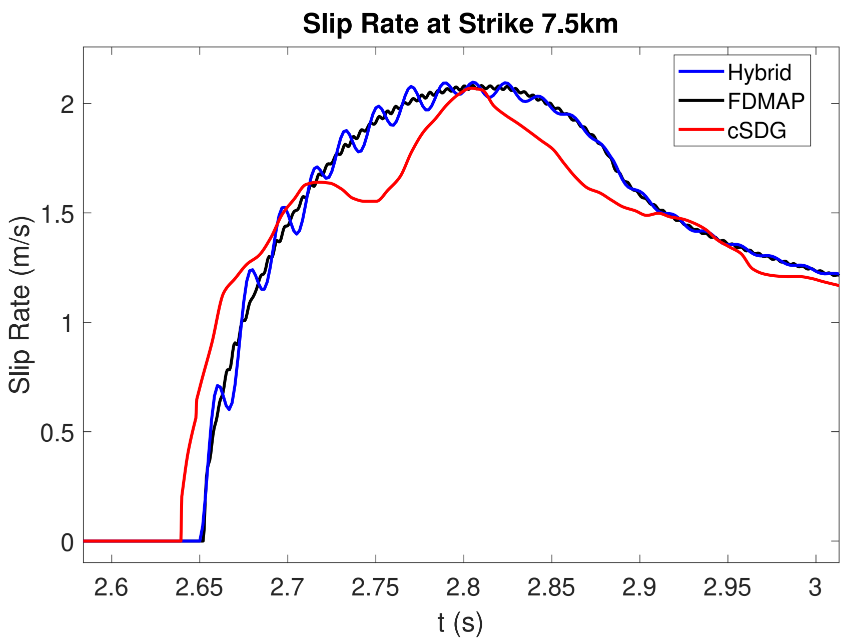


Figure 4.10: Slip-rate histories for TPV205-2D at $x^1 = 7.5$km by Hybrid, FDMAP, and cSDG models.

all spacetime inter-element boundaries, thereby eliminating spurious oscillations without resorting to filtering or artificial damping.[8]

Most other methods use some form of explicit time integration with stability constraints that impose restrictions on time-step size that grow increasingly severe with increasing polynomial order. cSDG models are not subject to stability constraints. Instead, the causality constraint acts locally and independently of polynomial order to limit the duration of individual patches. As a result, and in contrast to most other models, cSDG convergence rates and computational efficiency both improve with increasing polynomial order. Moreover, fine-grained and localized h-adaptive cSDG meshing procedures enable dynamic evolution of strongly-graded spacetime meshes capable of capturing fast-moving wavefronts and dynamic rupture propagation. This further boosts computational efficiency to enable models that bridge wider ranges of length and time scales.

Figure 4.10 compares the accuracy and resolution of slip-rate histories from the Hybrid, FDMAP, and cSDG solutions during a brief time interval in which the rupture tip passes the on-fault station point at $x^1 = 7.5$km. Spurious oscillations are evident in both the Hybrid and FDMAP solutions. The smaller amplitude and higher frequency of the FDMAP oscillations

[8]This statement is valid for the class of problems and the formulation considered in this work. Although models based on the present cSDG formulation are always conservative and unconditionally stable, they may generate Gibbs artifacts in problems where the exact solution contains weak shocks. However in these cases, the Gibbs artifacts are local and do not lead to global ringing as in many other methods.

reflect the more refined mesh used in that model. Although the oscillations cause relatively small slip-rate errors, the local deviations in slope, associated with differences in material acceleration across the fault, are significant in both models. In contrast, there is no evidence of spurious oscillations in the undamped and unfiltered cSDG solution.

Now consider the peak value of shear stress along the main fault as a function of time. The theoretical model predicts that as a rupture extends beyond the current rupture tip, the shear stress must reach and cannot excede the static strength, $\overline{\tau}$, as determined by the slip-weakening model. In practice, however, we find that this peak shear-stress value oscillates over the course of the simulation, as depicted in Figure 4.11 for the Hybrid and cSDG methods. Due to its high-resolution capabilities, the cSDG model's stress peaks are, overall, closer to the theoretical static strength value.

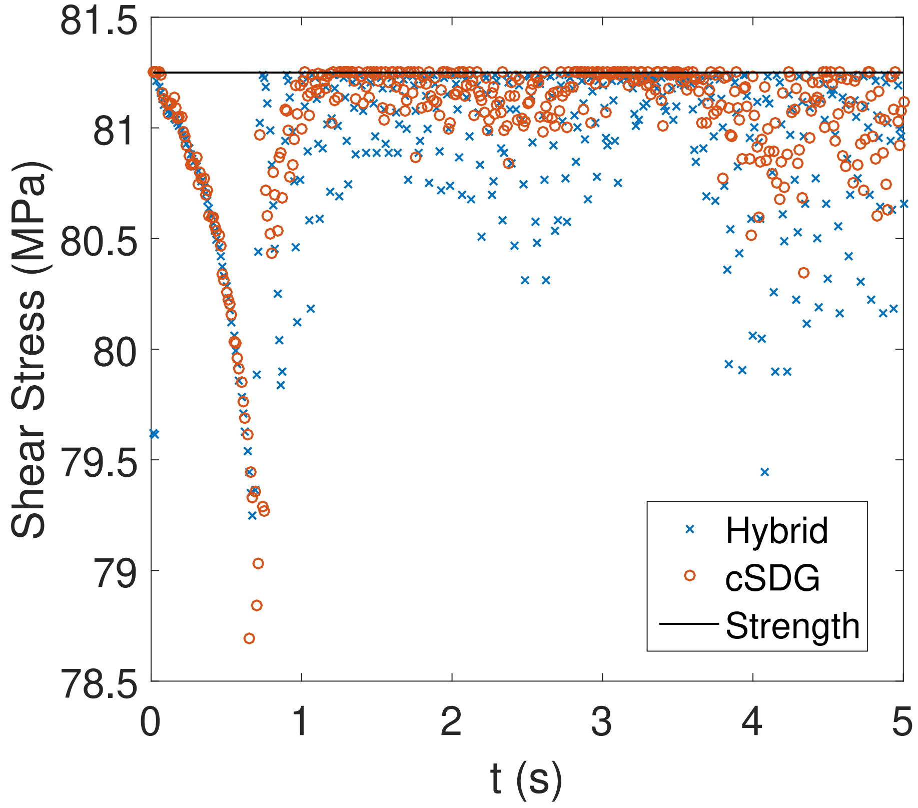


Figure 4.11: Time history of peak value of shear stress along the main fault for the Hybrid ($\times$) and cSDG ($\circ$) methods. The value of static strength, $\overline{\tau}$, is plotted as a horizontal solid line.

If we set aside the oscillations in the Hybrid and FDMAP solutions, we observe similarities and some differences between the three slip-rate histories in Fig. 4.10. All three solutions attain about the same peak slip-rate at about the same time, and the Hybrid and FDMAP solutions are in close agreement overall. However, the cSDG solution predicts earlier rupture arrival at the station point and resolves higher-frequency features with more local extrema than the others. There are also significant differences in the slip rate and "slip acceleration" — for example, in the interval, $2.7\text{s} < t < 2.9\text{s}$. The cSDG model's higher resolution is to be expected given its high polynomial order and stronger adaptive refinement compared with the other two methods.

Figure 4.12 presents a sequence of visualizations of the cSDG solution for TPV205-2D in which velocity magnitude is mapped to a synthetic height field to reveal wave structure and the dynamic part of strain energy density[9] is mapped to color, ranging from dark blue for zero energy density to red for peak values. The images in subfigures (a-b) depict the early stages of rupture propagation beyond the nucleation zone during which both tips accelerate and intensify. In subfigure (c), the left rupture tip intensifies and accelerates as it traverses the left zone of higher initial shear stress while the right tip weakens and decelerates as it crosses the right zone of lower initial shear stress. In subfigure (d), the left rupture tip exits the zone of increased initial stress carrying a more complex field while the right tip continues its slower transit of the reduced-stress heterogeneity. Subfigures (e,f) capture rupture arrest and reflection of the rupture-tip wave field at the left end of the fault while the right rupture tip accelerates toward the right end with a strengthening field. The cSDG model's high-order approximation and powerful adaptive meshing capabilities ensure that complex rupture dynamics and wave patterns are well resolved throughout the simulation.

4.4.2 Zero-Gap Variant of SCEC TPV14-2D Benchmark

Problem description

The TPV14-2D benchmark involves two vertical strike-slip faults, a main fault and a branch fault separated from the main fault by a vertical gap. A number of solutions are available in the SCEC repository for this non-intersecting configuration [80]. Here we focus on the zero-gap variant of TPV14-2D, previously analyzed by Pelties *et al.* [78] using the Arbitrary Derivative Discontinuous Gelerkin (ADER-DG) method, in which the main and branch faults intersect at an angle of 30° as depicted in Fig. 4.13a. The fault dimensions and other model parameters appear in Table 4.2, most of which are defined similarly to the parameters for TPV205-2D.

Due to the asymmetrical configuration and loading, the dynamic component of the normal traction field no longer vanishes, so the total normal-traction distribution varies

[9]We cannot visualize the total strain-energy-density field because the static initial strain field is unavailable in this model.

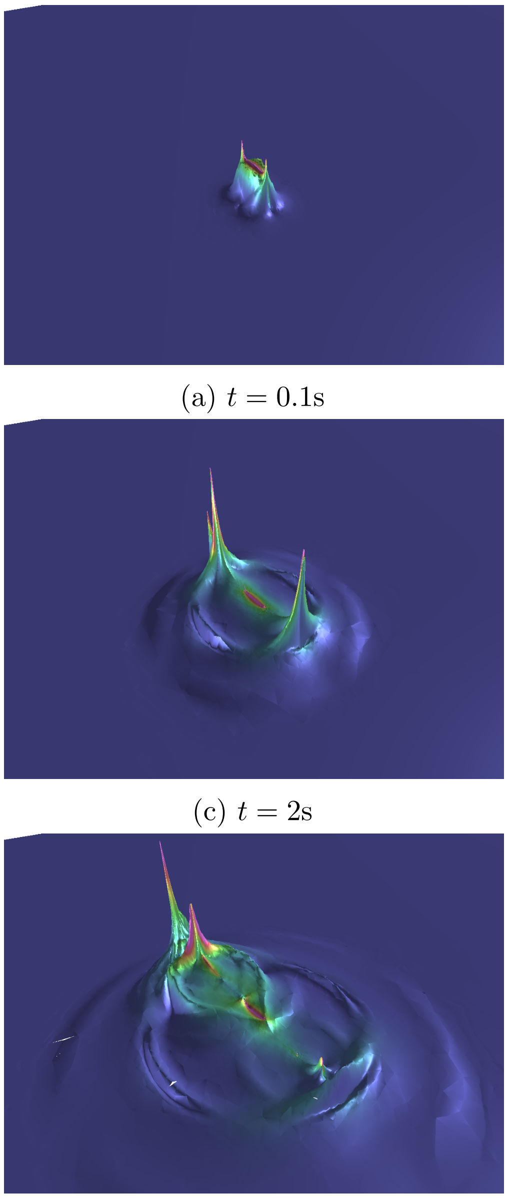
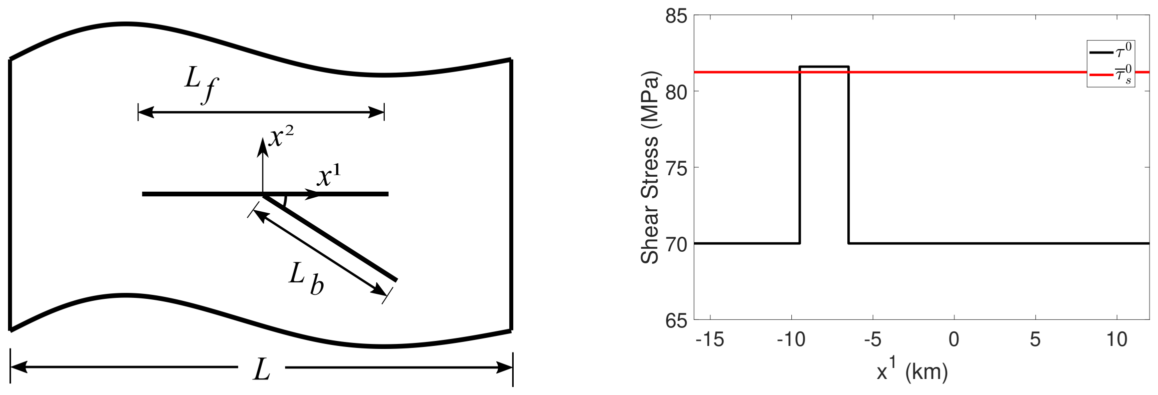

(a) $t = 0.1$s
(b) $t = 1$s
(c) $t = 2$s
(d) $t = 3$s
(e) $t = 4$s
(f) $t = 5$s

Figure 4.12: Visualization sequence for cSDG simulation of TPV205-2D with velocity magnitude and dynamic part of strain-energy density mapped, respectively, to height and color.



(a) (b)

Figure 4.13: Zero-gap variant of TPV14-2D. (a): Schematic of main and branch faults intersecting at 30° angle. (b): Distributions along the main fault of right-lateral static component of shear traction, τ^0, and static shear strength, $\overline{\tau}_{\mathrm{s}}^0$, at time $t = 0$.

with time. Accordingly, we write the static and dynamic shear resistances as $\overline{\tau}(t) = k_{\mathrm{s}}\sigma(t)$ and $\tilde{\tau}(t) = k_{\mathrm{d}}\sigma(t)$ in which $\sigma(t)$ satisfies $\sigma(0) = \sigma^0$. Thus, the initial static shear strength is $\overline{\tau}_{\mathrm{s}}^0 = k_{\mathrm{s}}\sigma^0$. The background value of right-lateral static shear tractions, $\tau^0 = \tau_{\mathrm{b}}^0$, holds everywhere on the fault system except in a 3m-wide nucleation zone, centered at $x^1 = -8$m on the main fault, within which $\tau^0 = \tau_{\mathrm{n}}^0$. As indicated in Fig. 4.13b, τ_{n}^0, slightly exceeds $\overline{\tau}_{\mathrm{s}}^0$, leading to instantaneous rupture across the nucleation zone at $t = 0$.

Verification and High-resolution modeling capabilities

We verify our method against the Aribtrary high-order discontinuous Galerkin (ADER-DG) method [78] – one of the few methods that is able to simulate the case without a gap at the fault junction point.

Figure 4.15 shows the slip and slip rate comparison at ±2.0km along-strike from the junction on the main fault and 2km along strike from the junction on the branch fault. Again we see that our method accurately captures all stages of rupture propagation along both the main and branch faults.

The cSDG method is able to simulate the TPV14-2D fault geometry, both with and without a 100m vertical gap between the main and branch faults at their junction point. We compare the slip, slip-rate, and shear stress solutions for the two cases in Figure 4.16 for a station point at 2km along-strike on the While the overall response is comparable for the two cases, the branch rupture is delayed and slightly diminished in the with-gap case in which the rupture-tip waves must traverse the gap before impinging on the branch fault.

Table 4.2: TPV14-2D model parameters

Parameter	Symbol	Value
Domain width (km)	L	100
Main-fault length (km)	L_f	28
Branch-fault length (km)	L_b	12
Dilatational wave speed (m/s)	c_d	6000
Shear wave speed (m/s)	c_s	3464
Mass density (kg/m^3)	ρ	2670
Slip-weakening distance (m)	δ_s	0.4
Static coefficient of friction	k_s	0.677
Dynamic coefficient of friction	k_d	0.525
Slip-weakening regularization parameter	ϵ	0.025
Uniform static compressive traction (MPa)	σ^0	120
Initial static shear strength (MPa)	$\overline{\tau}_\mathrm{s}^0$	81.24
Background static shear traction (MPa)	τ_b^0	70.0
Static shear traction in nucleation zone (MPa)	τ_n^0	81.6
Normal velocity error tolerance	$\kappa_\mathbf{v}^\mathrm{n}$	1×10^{-6}
Tangential velocity error tolerance	$\kappa_\mathbf{v}^\mathrm{t}$	1×10^{-6}
Normal stress error tolerance	$\kappa_\mathrm{n}^\mathbf{S}$	1×10^{-8}
Tangential stress error tolerance	$\kappa_\mathrm{t}^\mathbf{S}$	1×10^{-8}

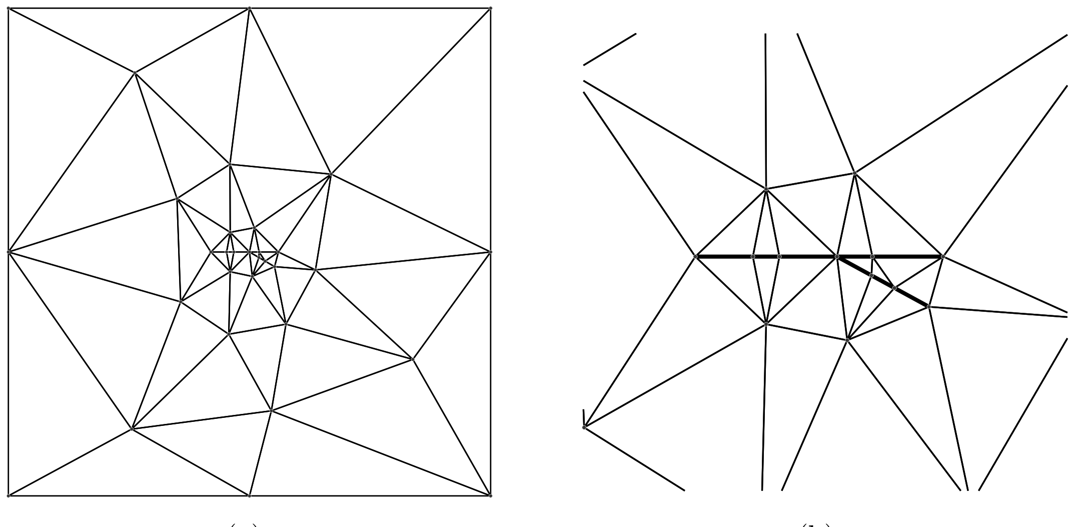
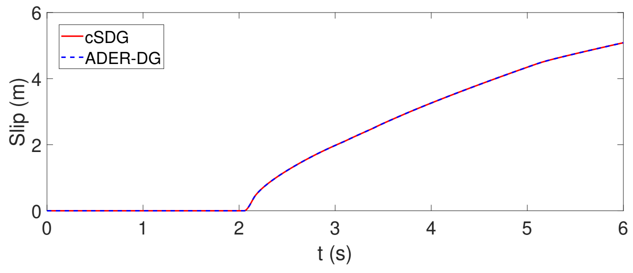
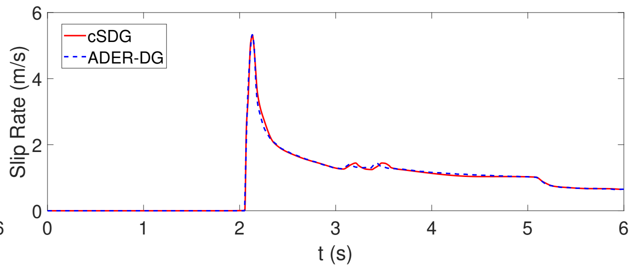

(a) (b)

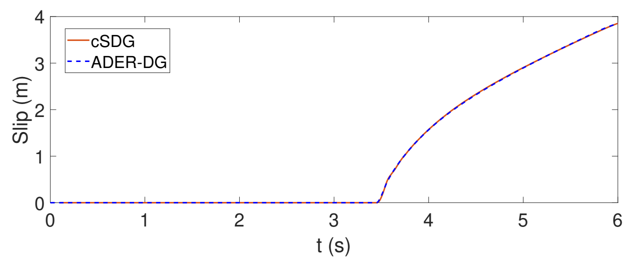
Figure 4.14: Initial front mesh for zero-gap version of TPV14-2D. (a) Overall mesh covering a 100km × 100km spatial domain with 32 vertices and 54 elements; (b) Detail of fault region.



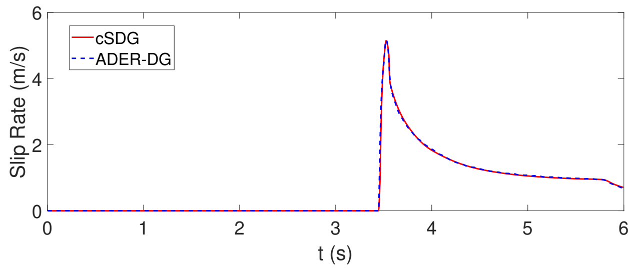
(a) Main-fault slip at $x^1 = -2$km



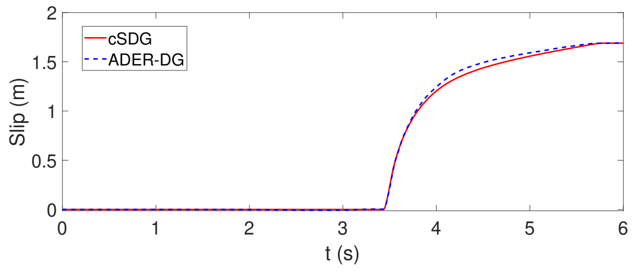
(b) Main-fault slip rate at $x^1 = -2$km



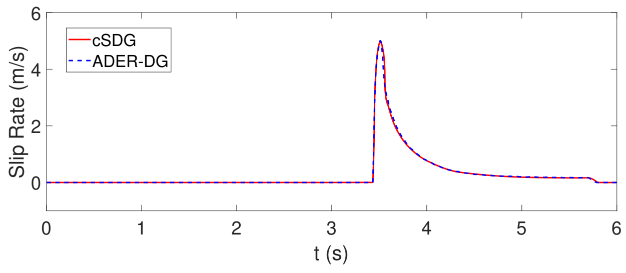
(c) Main-fault slip at $x^1 = 2$km



(d) Main-fault slip rate at $x^1 = 2$km

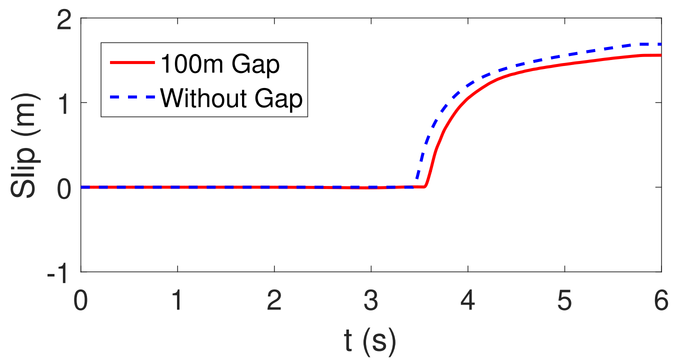


(e) Branch-fault slip, 2km from junction

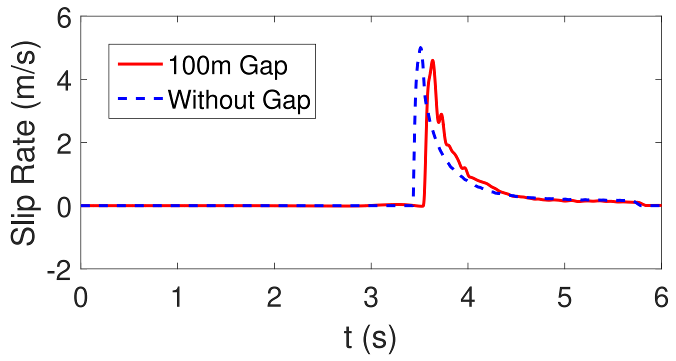


(f) Branch-fault slip rate, 2km from junction

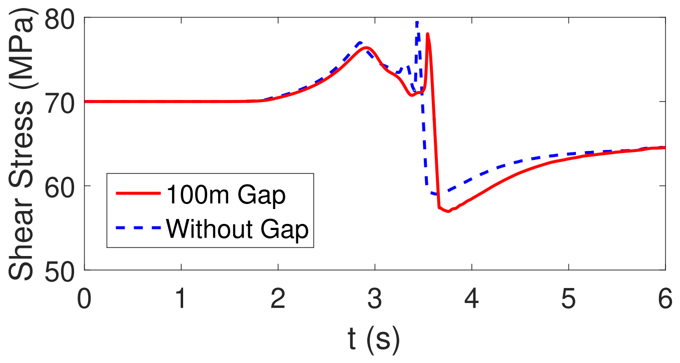
Figure 4.15: Slip and slip-rate histories for zero-gap variant of TPV14-2D at station points on main and branch faults by cSDG and ADER-DG methods.



(a)



(b)

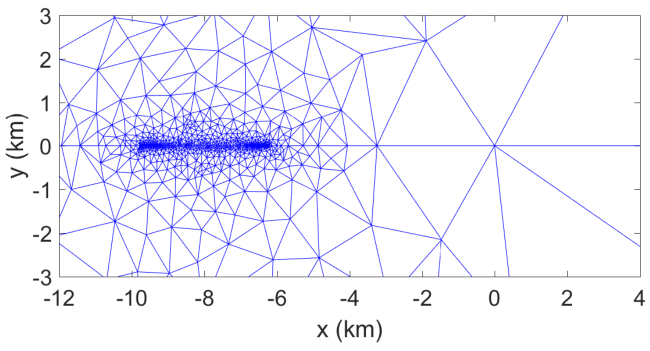


(c)

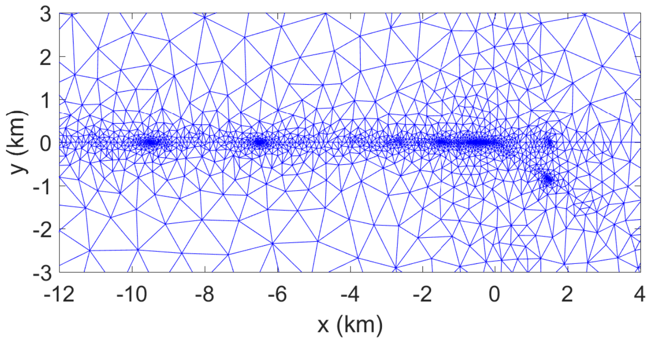
Figure 4.16: Comparison of TPV14-2D response with and without a 100m gap at the junction point. Plots of (a): slip, (b): slip rate, and (c): shear stress - recorded at a station point along the branch fault at 2km from the branch point.

The powerful fine-grained adaptivity inherent to our method is particularly well suited for branching fault systems such in the TPV14-2D benchmark. A direct advantage is that the computational cost for our simulations are directly focused in regions where it is needed. This produces computationally efficient meshes that allow for higher resolution and inclusion of intricate fault systems. While we have the capability to perform both $h-$ and $p-$refinement simultaneously on a patch level (since we allow for discontinuities across spacetime facets) we focus on the former for this work. The adaptive mesh refinement and coarsening used in our method is described briefly here, with a more complete description found in [18] and [15]. Adaptive operations are performed directly on the triangulation of the current front mesh. A local error indicator is computed for each element of a patch once it is solved and compared to a user-defined target value. If the error indicator on any element is too high compared to its target value, the patch is discarded and the element is marked for refinement. The facets of the current front that correspond to the marked elements are then refined. Compared to traditional global re-meshing scheme, the cSDG method only requires only the solution over the discarded element to be recomputed while all previously solved patches are unaffected due to causal structure of the spacetime mesh. On the other hand, if the error indicator in a given element is too small compared to the target value, the element is marked as coarsenable and the facets of the front are coarsened locally by undoing a single edge bisection. If none of the elements in a patch are marked for further adaptivity, the patch solution is stored and the local front is advanced. These refinement and coarsening operations produce nonconforming spacetime meshes which can be naturally accommodated by the discontinuous basis utilized in the cSDG formulation. This circumvents the need to project solution quantities between refinement steps - a source of numerical error and algorithmic complexity that impacts many other adaptive algorithms.

We consider the front mesh evolution for the TPV14-2D benchmark problem, presented in Figure 4.17, to illustrate the capabilities of the adaptive scheme. The front is a spacetime terrain with non-uniform time coordinates which is plotted here as a projection onto the $xy-$plane. Starting from the coarse initial mesh (*cf.* Figure 4.13c) with only 32 vertices, as the rupture propagates along the main fault, the adaptive algorithm concentrates refinement to regions around the rupture tips. Away from the rupture tips the mesh is rapidly coarsened. Figure 4.17b represents the front when rupture reaches the branch fault leading to intense refinement concentrated in the region of the junction point. Using a non-adaptive mesh of even $1m$ over the entire domain would require a prohibitively large number of elements. For the results presented in this paper, we routinely achieve meshes with element diameters under $0.1m$ within the rupture tip processes zone, giving an impressive maximum-to-minimum element ratio of roughly 10^6.



(a) $t \approx 0.1$s



(b) $t \approx 2.5$s

Figure 4.17: Spatial projections of front mesh details near the fault branch-point at two stages of the TPV14-2D simulation. The adaptive refinement scheme generates strongly graded front meshes - refining elements close to the rupture tips and coarsening ones further away.

Figure 4.18 presents a sequence of visualizations of the dynamic part of the cSDG solution where, as before, the velocity magnitude and strain energy density are mapped to the height and color fields respectively. The visualization depicts the acceleration of the initial rupture along the main fault (Figs. 4.18a-b) as indicated by the growing height field of the rupture tip regions. Immediately after reaching the branch point, the rupture tip decelerates and nucleates slip on the branch fault due to its favorable orientation (Figs. 4.18c-d).

Separation near the fault junction

Since the cSDG code is able to accurately model the triple-junction condition at the branch point, we are able to explicitly model the three regions around the branch as distinct domains. Based on the kinematic constraints and loading at the junction for the zero-gap case of TPV14-2D, we observe the following. When we apply right-lateral rupture along the fault system, we expect the bottom-right region of the branch to slip away from the junction and the bottom-left region to slip towards it. Thus, in order to have non-zero branch-fault slip

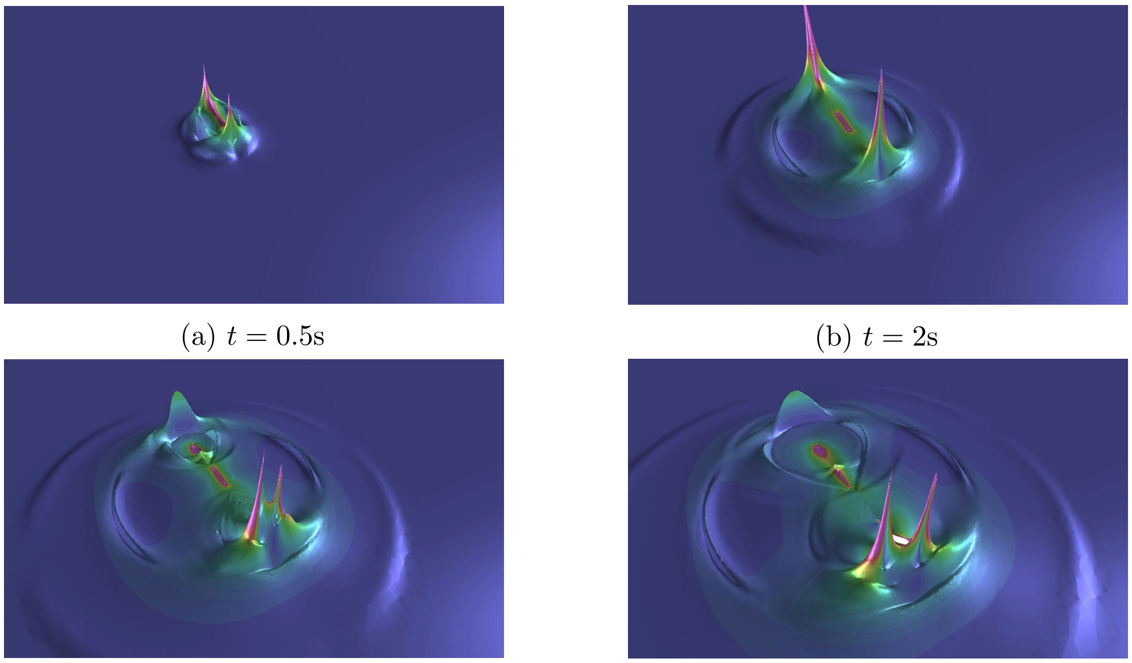

(a) $t = 0.5$s (b) $t = 2$s (c) $t = 3$s (d) $t = 5$s

Figure 4.18: Visualization sequence for cSDG solution of zero-gap TPV14-2D problem with the velocity magnitude and the dynamic part of strain-energy density mapped, respectively, to height and color.

at the junction, we must have local vertical separation on the main fault. Otherwise, the branch-fault slip must vanish at the junction point. We measure this separation for the TPV14-2D simulation, presented in Figure 4.19 where we find that separation accumulates as the rupture crosses the junction region.

Discovery of dipole-like feature at the fault junction

We were able to further push our adaptive capabilities to extreme levels of resolution to study fault dynamics in the vicinity of the junction point for the case of zero gap. Although our simulations revealed significant additional high-frequency detail, they agree with previous results that showed zero slip on the branch fault at the junction point, a kinematic consequence of zero separation on the main fault at the junction point. However, we observed a dipole-like feature in the main-fault normal stress as depicted in 4.20a; a feature that previous solutions are far too coarse to resolve. Combining tight adaptive tolerances to generate extreme spacetime mesh refinement and high-order spacetime bases ($p = 12$), we are able to achieve maximum resolutions in the $1 - 10mm$ range in the area of interest. The resolution is defined here as the distance between the junction point and the closest quadrature point in the surrounding elements. This is clearly observed in Figure 4.20b where we find the peak value

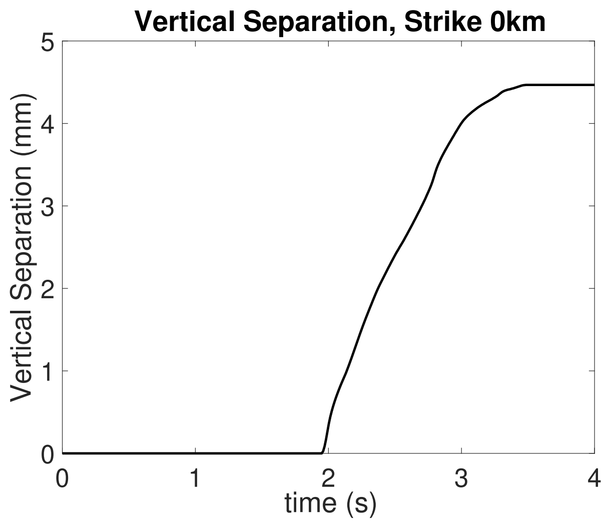


Figure 4.19: Vertical separation measured at the branch point for TPV14-2D between the two sides of the main fault.

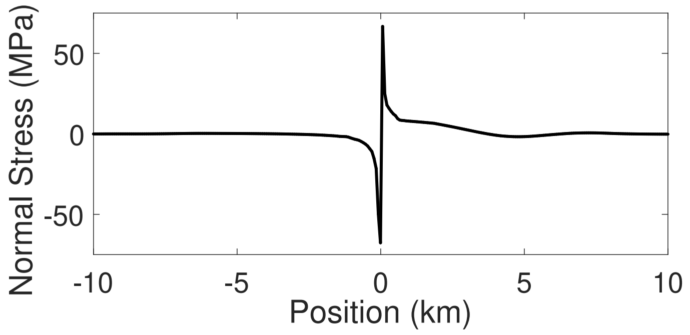
of the normal stress nearly overcomes the ambient compression of $\sigma^0 = 120MPa$.

Modest modifications of the model parameters led to solutions in which the dipole overcomes the ambient compression to produce a zone of separation on the main fault that permits significant non-zero slip on the branch fault at the junction fault. To our knowledge, this is the first description and simulation of this alternative mechanism of main–fault/branch-fault interaction. We note that accurate resolution of the peak dynamic normal stress due to the dipole-like response requires spatial resolution at scales much smaller than the scale of the rupture process zone which, in turn, is much smaller than the smallest scales resolved by most seismic rupture simulations.

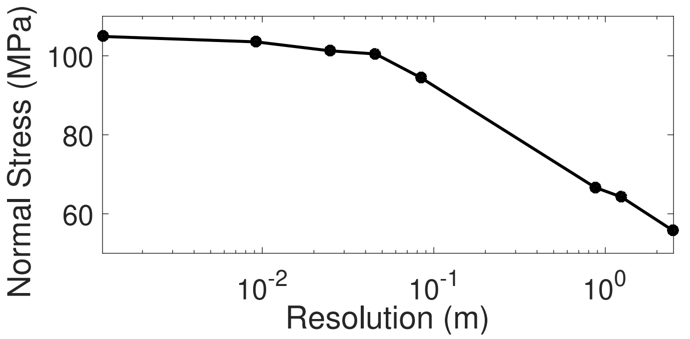
4.4.3 Modeling Off-fault Damage

Problem description

Many faults suffer off-fault damage in the form of complex networks of smaller-scale faults due to previous earthquakes. The experiments by Biegel, Sammis and Rosakis [82] in fracture-damaged Homalite suggest off-fault damage can significantly slow rupture speeds as compared to an undamaged fault system. The experimental sample used in the experiments are presented in Figure 4.21(a) where the damaged region extends on either side of the main fault. Many numerical codes model the damaged zone by smearing out the network of small-scale faults with a homogenized region with different material properties than that of the bulk material [83, 84]. Given the adaptive capabilities of the cSDG method, we are able to directly capture the side faults, depicted in Figure 4.21(b). We model the side faults



(a)



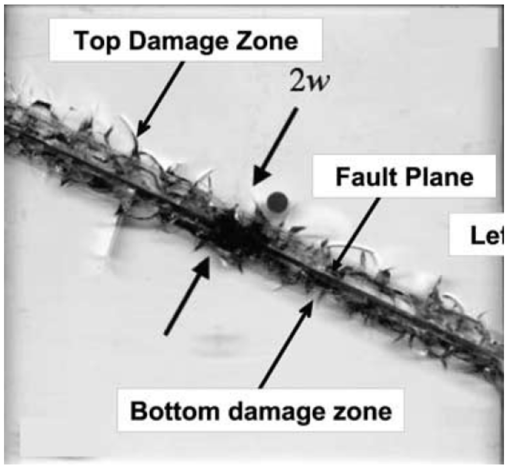
(b)

Figure 4.20: (a): Normal dynamic stress profile along the main fault for TPV14-2D depicting the presence of a stress-dipole at the junction point. (b): Increase in peak normal dynamic stress at the junction point with reduction in element resolution close to the junction point.

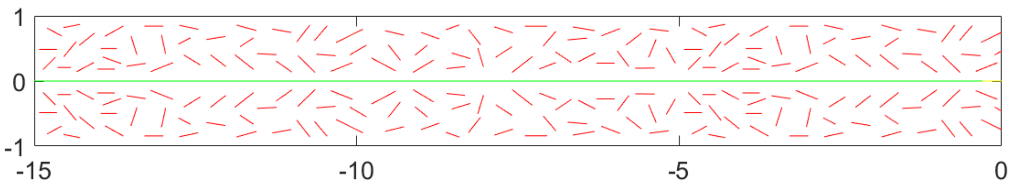
as discrete, non-overlapping segments with random orientation with an average length of $300m$ in a damaged region measuring $15km \times 2km$. We utilize symmetry across the y-axis to simplify the mesh generation.

Parallel set-up

Due to the inclusion of smaller-scale fault systems, this model exacerbates the multi-scale issue inherent to modeling seismic problems. The problem serves as an excellent demonstration of the potential speed-ups offered by our parallel–adaptive cSDG code described in Chapter 3. We utilize the simpler Xu and Needleman traction separation relationship implemented for the cSDG method [85] due to certain constraints on the parallel code at the time. The problem is run on 2 compute nodes on the HAL cluster, each utilizing 40 CPU cores. The final run-time for the simulation was approximately 6 hours, whereas a typical serial run with similar parameters would take upwards of 2 weeks.



(a)

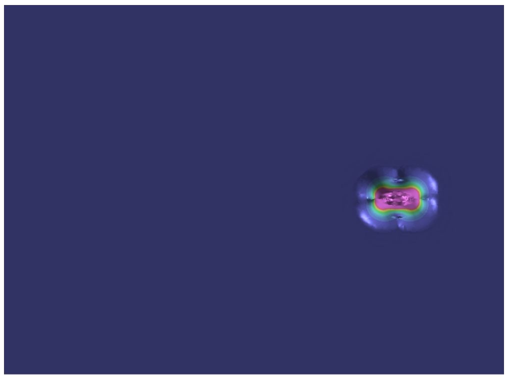


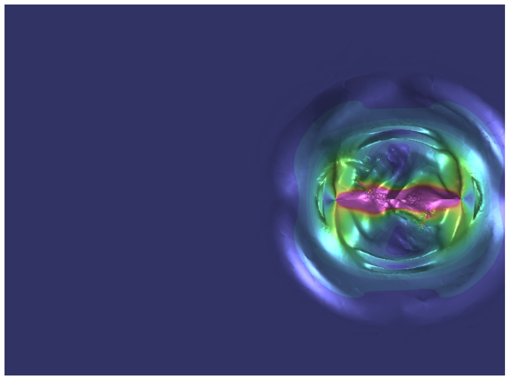
(b)

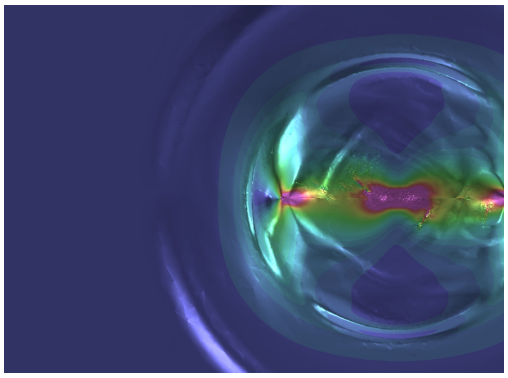
Figure 4.21: Off-fault damage model adapted from experiments by [82]. (a) Network of fracture-damaged side faults in Homalite with a half-width $w = 0.5$cm; (b) Initial front mesh generated to model off-fault damage with a random distribution of small-scale side faults.

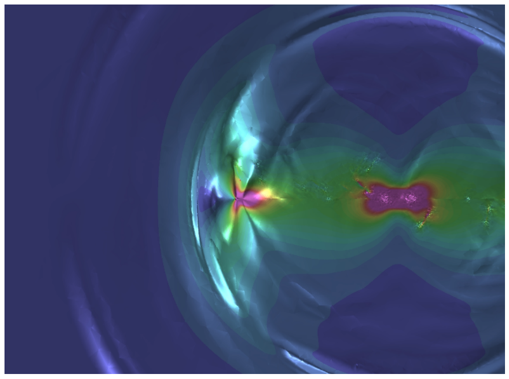
Visualization

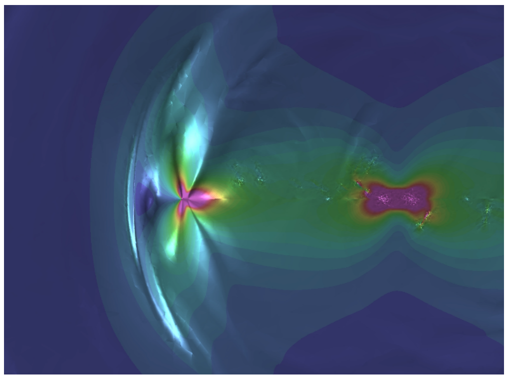
Figure 4.22 presents a time sequence of visualizations of the dynamic part of the cSDG solution field. As before, the velocity magnitude is mapped to the height field and the strain energy density is mapped to the color field. After the initial nucleation of the rupture (Figure 4.22a), the resulting stress waves interact with the side-faults, activating and generating reflections along them (Figs. 4.22b-d). The multiple interacting wave fronts are all well resolved due to the adaptive meshing of the cSDG method. As the rupture moves past the damaged region, side faults with favorable orientations are activated (indicated by the presence of discolored artifacts) which causes slowing of the rupture along the main fault, as predicted by the experimental results (Fig. 4.22e-f).


(a) $t = 0.1$s


(b) $t = 0.5$s


(c) $t = 1$s


(d) $t = 1.5$s


(e) $t = 2$s

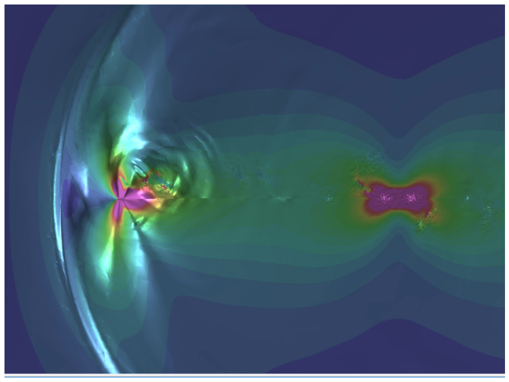

(f) $t = 2.5$s

Figure 4.22: Top-view visualization sequence of elastic response for the randomly oriented fault network by cSDG method. Strain energy density and velocity magnitude are mapped, respectively, to color and height fields.

4.5 Chapter Summary

We extended the cSDG method to model multi-scale earthquake rupture problems. The unstructured and non-conforming mesh discretizations enabled by the cSDG method enables us to capture the fine details of earthquake rupture dynamics. We combined a dynamic contact model and a slip-weakening friction model with the cSDG model for linear elastodynamics to build the dynamic rupture model for fault dynamics that is unconditionally stable, conservative of linear and angular momentum to machine precision over every spacetime cell, and linear in computational complexity. These features have been leveraged to perform numerical verification against various benchmark problems in the SCEC database where we find that the cSDG rupture model yields solutions that are free of oscillations and capable of resolving features, such as stress dipoles and separation zones at branch-fault intersections, that would be prohibitively expensive with other non-adaptive methods. Finally, we demonstrated the potential power of parallel–adaptive cSDG implementations in a problem where we model off-fault damage explicitly as a network of small-scale faults in lieu of an elastoplastic smeared-damage models.

Chapter 5

Conclusions and Continuing Research

5.1 Conclusions

The causal Spacetime Discontinuous Galerkin method for hyperbolic systems has been extensively developed, primarily in serial form and in problems cast in two spatial dimension and time. Its fundamental features, element-wise basis functions and unstructured spacetime meshes that conform to the target system's characteristic structure, support asynchronous advancing-front solution schemes in which adaptive spacetime meshing and implicit finite element solution operations localize to the same patch-wise granularity. This property supports highly dynamic h-adaptive cSDG solvers with unconditional stability and linear computational complexity in the number of spacetime patches. Individual patch durations are determined by the causality constraint according to the local wave speeds and mesh geometry. Hence, in contrast to explicit methods in which the limiting stable time-step size decreases with increasing polynomial order, allowable patch durations are independent of polynomial order. Moreover, the causality constraint allows cSDG solutions to advance asynchronously without extrapolating solutions on lagging adjacent elements, as is required in asynchronous Local Time Stepping schemes. cSDG jump conditions, defined relative to Riemann solutions, maintain the system's characteristic structure across inter-element boundaries. Weak cSDG solutions satisfy conservation/balance laws to within machine precision over every spacetime element, and there is no need for filtering or artificial dissipation to suppress spurious grid-scale oscillations.

This unique feature set qualifies the cSDG method as an attractive candidate for new science and engineering applications and for advancing the state of the art in high-performance hyperbolic solvers. Accordingly, the original contributions of the research reported in this document are grouped under two topics: the first scalable framework for implementing parallel–adaptive cSDG solvers on distributed platforms and a new application of cSDG technology to multi-scale models of seismic rupture dynamics, as described in Chapters 3 and 4 and summarized below.

The design of effective parallel–adaptive hyperbolic solvers suitable for implementation on large-scale computing platforms with distributed architectures and GPU accelerators is an area of active research that presents multiple challenging problems. First, a highly dynamic form of adaptive modeling is required to capture traveling wavefronts as well as, for example, dynamically propagating cracks and ruptures. Dynamic cSDG h-adaptive meshing easily meets this requirement, but conventional methods for load and data balancing can not keep up. In particular, it is difficult or impossible to recompute traditional domain decompositions fast enough and often enough to keep up with cSDG h-adaptive meshing. Chapter 3 presents a scalable, distributed parallel–adaptive framework for the cSDG method that is asynchronous and virtually barrier and lock-free. We freely assign front-mesh data — our only global data structure — to distributed server processes and freely assign asynchronous patch-solution tasks to solver processes, both without regard to spatial coherence. In effect, we sacrifice data locality for the freedom to develop more effective schemes for load and data balancing, such as the break-through probabilistic balancing schemes that are criical to ParaView's scaling performance.

The new framework implements a task-based model of asynchronous parallel execution wherein an MPI-based server-client subsystem manages the distributed front-mesh data and feeds fragments of the front-mesh to asynchronous processes that perform spacetime patch construction, finite element patch solution, adaptive mesh refinement, as well as data and load balancing as embarrassingly parallel operations in local memory at a common patch-level granularity. We demonstrated that standard latency-hiding techniques in the server-client subsystem are sufficient to overcome the increased communication costs due to loss of data locality. The resulting parallel–adaptive cSDG code achieved excellent scaling efficiency on a distributed cluster.

Chapter 4 presents an application of the h-adaptive cSDG method to direct numerical simulation of multi-scale earthquake dynamics. We combine a dynamic contact model and a slip-weakening friction model with a cSDG implementation of linear elastodynamics to consruct a rupture dynamics model that is unconditionally stable, conservative of linear and angular momentum over every spacetime cell, and linear in computational complexity. The cSDG method's unstructured and non-conforming adaptive spacetime meshes capture fine details of earthquake rupture dynamics. Numerical studies using SCEC benchmark problems verify that the cSDG rupture model generates solutions that are free of spurious oscillations and capable of resolving extremely small-scale features that would be prohibitively expensive for non-adaptive methods to resolve. One study led to the discovery of a dipole-like feature that can cause separation at branch-fault/main-fault junctions. Finally, we demonstrated the potential power of parallel–adaptive cSDG implementations in a problem where we model off-fault damage explicitly as a network of small-scale faults in lieu of an elastoplastic smeared-damage models.

5.2 Continuing Research

Although algorithms for non-adaptive causal meshing in three spatial dimensions and time are available, robust algorithms for adaptive causal meshing in three-dimensions and time have been elusive. Thus, cSDG models have been limited to one and two spatial dimensions — a significant restriction in many science and engineering applications. A recent breakthrough at Illinois [86] has produced the first dimension-independent algorithms for adaptive causal meshing in arbitrary spatial dimensions. The parallel–adaptive framework described in this dissertation and causal adaptive meshing in up to three spatial dimensions have been combined in a new cSDG code called *ParaSDG*. This represents a significant milestone in the development of cSDG technology and opens up a broader range of applications. For example, ParaSDG will allow us to run the full suite of three-dimensional SCEC benchmarks as well as other popular problems in the seismic community [65], and we are considering new applications involving tribology and general relativity in astrophysics.

Planned enhancements of the ParaSDG framework include the use of Trefftz basis functions to eliminate integrals over spacetime element interiors, incorporation of gpus within the parallel framework, *hp*-adaptive models, and an improved variational formulation for elastodynamics. We have ported ParaSDG to the new Delta cluster at the National Center for Supercomputing Applications. Delta provides a significantly larger system on which to benchmark and tune ParaSDG's scalability. A number of optimizations are planned to address known issues and enhance ParaSDG's baseline performance and scalability. For example, we have identified contention for shared-memory locks between MPI system processes and cSDG solver processes requesting dynamic memory allocations as the major cause of less-than-ideal scaling in our parallel–adaptive codes. We have mitigated this problem to some extent, but we believe that it's possible to eliminate all dynamic memory allocation requests after a solver process is initialized, and we expect this to bring ParaSDG much closer to ideal scaling.

References

[1] Reza Abedi, Robert B Haber, and Boris Petracovici. “A spacetime discontinuous Galerkin method for elastodynamics with element-level balance of linear momentum”. In: Comput. Methods Appl. Mech. Eng. 195.25-28 (2006), pp. 3247–3273.

[2] Reza Abedi and Robert B. Haber. “Riemann solutions and spacetime discontinuous Galerkin method for linear elastodynamic contact”. In: Comput. Methods Appl. Mech. Eng. 270 (2014), pp. 150–177. ISSN: 00457825.

[3] Reza Abedi and Robert B Haber. “Spacetime simulation of dynamic fracture with crack closure and frictional sliding”. In: Advanced Modeling and Simulation in Engineering Sciences 5.1 (2018), pp. 1–22.

[4] Leslie G Valiant. “A bridging model for parallel computation”. In: Comm. of ACM 33.8 (1990), pp. 103–111.

[5] Gregory A Lyzenga, Arthur Raefsky, and Bradford H Hager. “Finite elements and the method of conjugate gradients on a concurrent processor”. In: (1984).

[6] James G Malone. “Automated mesh decomposition and concurrent finite element analysis for hypercube multiprocessor computers”. In: Computer methods in applied mechanics and engineering 70.1 (1988), pp. 27–58.

[7] William H Reed and Thomas R Hill. “Triangular mesh methods for the neutron transport equation”. In: (1973).

[8] Bernardo Cockburn, George E Karniadakis, and Chi-Wang Shu. “The development of discontinuous Galerkin methods”. In: Discontinuous Galerkin Methods. Springer, 2000, pp. 3–50.

[9] Thomas JR Hughes and Gregory M Hulbert. “Space-time finite element methods for elastodynamics: formulations and error estimates”. In: Computer methods in applied mechanics and engineering 66.3 (1988), pp. 339–363.

[10] Nils-Erik Wiberg, Lingfu Zeng, and Xiangdong Li. “Error estimation and adaptivity in elastodynamics”. In: Computer Methods in Applied Mechanics and Engineering 101.1-3 (1992), pp. 369–395.

[11] Xiang Dong Li and N-E Wiberg. “Structural dynamic analysis by a time-discontinuous Galerkin finite element method”. In: International Journal for Numerical Methods in Engineering 39.12 (1996), pp. 2131–2152.

[12] Gerard R Richter. “An explicit finite element method for the wave equation”. In: Applied Numerical Mathematics 16.1-2 (1994), pp. 65–80.

[13] Lin Yin et al. “A space-time discontinuous Galerkin method for elastodynamic analysis”. In: Discontinuous Galerkin Methods. Springer, 2000, pp. 459–464.

[14] Reza Abedi et al. “Adaptive discontinuous galerkin method for elastodynamics on unstructured spacetime grids”. In: Proc. XXI International Congress of Theoretical and Applied Mechanics (ICTAM). 2004.

[15] Reza Abedi et al. “An h-adaptive spacetime-discontinuous Galerkin method for linear elastodynamics”. In: European Journal of Computational Mechanics 15.6 (2006), pp. 619–642.

[16] J. Erickson et al. “Building spacetime meshes over arbitrary spatial domains”. In: Engineering with Computers 20.4 (2005), pp. 342–353.

[17] A. Üngör and A. Sheffer. “Pitching tents in space-time: Mesh generation for discontinuous Galerkin method”. In: International Journal of Foundations of Computer Science 13.2 (2002), pp. 201–221.

[18] Reza Abedi et al. “Spacetime meshing with adaptive refinement and coarsening”. In: Proc. Annu. Symp. Comput. Geom. ACM. 2004, pp. 300–309.

[19] Reza Abedi, Boris Petracovici, and Robert B Haber. “A space–time discontinuous Galerkin method for linearized elastodynamics with element-wise momentum balance”. In: Computer Methods in Applied Mechanics and Engineering 195.25-28 (2006), pp. 3247–3273.

[20] S. T. Miller et al. "Multi-field spacetime discontinuous Galerkin methods for linearized elastodynamics". In: Computer Methods in Applied Mechanics and Engineering 199 (2009), pp. 34–47.

[21] Reza Abedi et al. "An adaptive spacetime discontinuous Galerkin method for cohesive models of elastodynamic fracture". In: International journal for numerical methods in engineering 81.10 (2010), pp. 1207–1241.

[22] Reza Abedi and Philip L Clarke. "A computational approach to model dynamic contact and fracture mode transitions in rock". In: Computers and Geotechnics 109 (2019), pp. 248–271.

[23] Omid Omidi, Reza Abedi, and Saeid Enayatpour. "An adaptive meshing approach to capture hydraulic fracturing". In: 49th US Rock Mechanics/Geomechanics Symposium. OnePetro. 2015.

[24] Philip L Clarke and Reza Abedi. "Modeling the connectivity and intersection of hydraulically loaded cracks with in situ fractures in rock". In: International journal for numerical and analytical methods in geomechanics 42.14 (2018), pp. 1592–1623.

[25] Raj Kumar Pal et al. "Adaptive spacetime discontinuous Galerkin method for hyperbolic advection–diffusion with a non-negativity constraint". In: International Journal for Numerical Methods in Engineering 105.13 (2016), pp. 963–989.

[26] Philip Clarke et al. "Space-angle discontinuous Galerkin method for plane-parallel radiative transfer equation". In: Journal of Quantitative Spectroscopy and Radiative Transfer 233 (2019), pp. 87–98.

[27] Reza Abedi and Saba Mudaliar. "An asynchronous spacetime discontinuous Galerkin finite element method for time domain electromagnetics". In: Journal of Computational Physics 351 (2017), pp. 121–144.

[28] Kenneth Eriksson, Claes Johnson, and Anders Logg. "Explicit time-stepping for stiff ODEs". In: SIAM Journal on Scientific Computing 25.4 (2004), pp. 1142–1157.

[29] R. Abedi and R. Haber. "Riemann solutions and spacetime discontinuous Galerkin method for linear elastodynamic contact". In: Comput. Methods Appl. Mech. Engrg. 270 (2014), pp. 150–177.

[30] Shripad Vidyadhar Thite. “Spacetime meshing for discontinuous Galerkin methods”. PhD thesis. University of Illinois at Urbana-Champaign, 2005.

[31] R. Abedi et al. “Spacetime meshing with adaptive refinement and coarsening”. In: Proceedings of the Annual Symposium on Computational Geometry (2004), pp. 300–309.

[32] S. Thite. “Adaptive spacetime meshing for discontinuous Galerkin methods”. In: Computational Geometry: Theory and Applications 42.1 (2009), pp. 20–44.

[33] Amit Madhukar. “Asynchronous parallel solver for hyperbolic problems via the Spacetime Discontinuous Galerkin method”. MA thesis. University of Illinois at Urbana-Champaign, 2016.

[34] James D Teresco, Karen D Devine, and Joseph E Flaherty. “Partitioning and dynamic load balancing for the numerical solution of partial differential equations”. In: Numerical solution of partial differential equations on parallel computers. Springer, 2006, pp. 55–88.

[35] Jianjun Chen et al. “Scalable generation of large-scale unstructured meshes by a novel domain decomposition approach”. In: Advances in Engineering Software 121 (2018), pp. 131–146.

[36] J.G. Malone. “Automated mesh decomposition and concurrent finite element analysis for hypercube multiprocessor computers”. In: Computer Methods in Applied Mechanics and Engineering 70.1 (1988), pp. 27–58.

[37] J.-C. Luo and M.B. Friedman. “A parallel computational model for the finite element method on a memory-sharing multiprocessor computer”. In: Computer Methods in Applied Mechanics and Engineering 84.2 (1990), pp. 193–209.

[38] D. Goehlich, L. Komzsik, and R.E. Fulton. “Application of a parallel equation solver to static fem problems”. In: Computers and Structures 31.2 (1989), pp. 121–129.

[39] G. Karypis and V. Kumar. “A fast and high quality multilevel scheme for partitioning irregular graphs”. In: SIAM Journal on Scientific Computing 20.1 (1998), pp. 359–392.

[40] K.T. Danielson et al. “Parallel computation of meshless methods for explicit dynamic analysis”. In: International Journal for Numerical Methods in Engineering 47.7 (2000), pp. 1323–1341.

[41] C. Farhat and F. X. Roux. “A method of finite element tearing and interconnecting and its parallel solution algorithm”. In: International Journal for Numerical Methods in Engineering 32.6 (1991), pp. 1205–1227.

[42] C. Farhat et al. “FETI-DP: A dual-primal unified FETI method part I: A faster alternative to the two-level FETI method”. In: International Journal for Numerical Methods in Engineering 50.7 (2001), pp. 1523–1544.

[43] B. Cockburn and C.-W. Shu. “The Runge-Kutta Discontinuous Galerkin Method for Conservation Laws V: Multidimensional Systems”. In: Journal of Computational Physics 141.2 (1998), pp. 199–224.

[44] F. Bassi and S. Rebay. “High-order accurate discontinuous finite element solution of the 2D Euler equations”. In: Journal of Computational Physics 138.2 (1997), pp. 251–285.

[45] H.L. Atkins and C.-W. Shu. “Quadrature-free implementation of discontinuous Galerkin method for hyperbolic equations”. In: AIAA Journal 36.5 (1998), pp. 775–782.

[46] Y. Xia et al. “A set of parallel, implicit methods for a reconstructed discontinuous Galerkin method for compressible flows on 3D hybrid grids”. In: Computers and Fluids 98 (2014), pp. 134–151.

[47] H. Luo et al. “A parallel, reconstructed discontinuous galerkin method for the compressible flows on arbitrary grids”. In: Communications in Computational Physics 9.2 (2011), pp. 363–389.

[48] Karen D Devine et al. “A massively parallel adaptive finite element method with dynamic load balancing”. In: Proceedings of the 1993 ACM/IEEE conference on Supercomputing. 1993, pp. 2–11.

[49] Joseph E Flaherty et al. “Distributed octree data structures and local refinement method for the parallel solution of three-dimensional conservation laws”. In: Grid Generation and Adaptive Algorithms. Springer, 1999, pp. 113–134.

[50] Yidong Xia et al. “A set of parallel, implicit methods for a reconstructed discontinuous Galerkin method for compressible flows on 3D hybrid grids”. In: Computers & Fluids 98 (2014), pp. 134–151.

[51] L Prince Raj et al. “A super-parallel mixed explicit discontinuous Galerkin method for the second-order Boltzmann-based constitutive models of rarefied and microscale gases”. In: Computers & Fluids 157 (2017), pp. 146–163.

[52] Nico Krais et al. "FLEXI: A high order discontinuous Galerkin framework for hyperbolic–parabolic conservation laws". In: Computers & Mathematics with Applications 81 (2021), pp. 186–219.

[53] Laxmikant V Kale and Sanjeev Krishnan. "Charm++: Parallel programming with message-driven objects". In: Parallel programming using C+ 1 (1996), pp. 175–213.

[54] Kartik Marwah. "Numerical simulation of fragmentation and spalling and parallelization of a spacetime finite element code". MA thesis. University of Illinois at Urbana-Champaign, 2014.

[55] Robert M Metcalfe and David R Boggs. "Ethernet: Distributed packet switching for local computer networks". In: Communications of the ACM 19.7 (1976), pp. 395–404.

[56] Petra Berenbrink et al. "Balanced allocations: The heavily loaded case". In: SIAM J. Comp. 35.6 (2006), pp. 1350–1385.

[57] Volodymyr Kindratenko et al. "Hal: Computer system for scalable deep learning". In: Practice and Experience in Advanced Research Computing. 2020, pp. 41–48.

[58] François Broquedis et al. "hwloc: A generic framework for managing hardware affinities in HPC applications". In: 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. IEEE. 2010, pp. 180–186.

[59] Sameer S Shende and Allen D Malony. "The TAU parallel performance system". In: The International Journal of High Performance Computing Applications 20.2 (2006), pp. 287–311.

[60] Emery D Berger et al. "Hoard: A scalable memory allocator for multithreaded applications". In: ACM Sigplan Notices 35.11 (2000), pp. 117–128.

[61] Yossi Azar et al. "Balanced allocations". In: Proc. ACM symposium theory of computing. 1994, pp. 593–602.

[62] Frederick M Chester and Judith S Chester. "Ultracataclasite structure and friction processes of the Punchbowl fault, San Andreas system, California". In: Tectonophysics 295.1-2 (1998), pp. 199–221.

[63] DR Faulkner et al. "A review of recent developments concerning the structure, mechanics and fluid flow properties of fault zones". In: Journal of Structural Geology 32.11 (2010), pp. 1557–1575.

[64] H Igel et al. "Simulation of seismic wave propagation in media with complex geometries". In: AGU Fall Meeting Abstracts. Vol. 2011. 2011, S21B–2173.

[65] Babak Poursartip, Arash Fathi, and John L Tassoulas. “Large-scale simulation of seismic wave motion: A review”. In: Soil Dynamics and Earthquake Engineering 129 (2020), p. 105909.

[66] Daniel Peter et al. “Forward and adjoint simulations of seismic wave propagation on fully unstructured hexahedral meshes”. In: Geophysical Journal International 186.2 (2011), pp. 721–739.

[67] Alexander Heinecke et al. “Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers”. In: Proc. Int. Conference for High Performance Computing. IEEE. 2014, pp. 3–14.

[68] Carsten Uphoff et al. “Extreme scale multi-physics simulations of the tsunamigenic 2004 sumatra megathrust earthquake”. In: Proc. Int. Conference for High Performance Computing. 2017, pp. 1–16.

[69] Xiao Ma et al. “A hybrid finite element-spectral boundary integral approach: Applications to dynamic rupture modeling in unbounded domains”. In: Int. J. Numer. Anal. Methods Geomech. 43.1 (2019), pp. 317–338. ISSN: 10969853.

[70] Carsten Burstedde et al. “ALPS: A framework for parallel adaptive PDE solution”. In: Journal of Physics: Conference Series. Vol. 180. IOP Publishing. 2009, p. 012009.

[71] Martin Kronbichler, Timo Heister, and Wolfgang Bangerth. “High accuracy mantle convection simulation through modern numerical methods”. In: Geophysical Journal International 191.1 (2012), pp. 12–29.

[72] Michael Dumbser, Martin Käser, and Eleuterio F Toro. “An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes-V. Local time stepping and p-adaptivity”. In: Geophysical Journal International 171.2 (2007), pp. 695–717.

[73] Alexander Breuer, Alexander Heinecke, and Michael Bader. “Petascale local time stepping for the ADER-DG finite element method”. In: 2016 IEEE international parallel and distributed processing symposium (IPDPS). IEEE. 2016, pp. 854–863.

[74] M Benjemaa et al. “Dynamic non-planar crack rupture by a finite volume method”. In: Geophysical Journal International 171.1 (2007), pp. 271–285.

[75] Josep de la Puente, J-P Ampuero, and Martin Käser. “Dynamic rupture modeling on unstructured meshes using a discontinuous Galerkin method”. In: Journal of Geophysical Research: Solid Earth 114.B10 (2009).

[76] Josué Tago et al. “A 3D hp-adaptive discontinuous Galerkin method for modeling earthquake dynamics”. In: Journal of Geophysical Research: Solid Earth 117.B9 (2012).

[77] Christian Pelties et al. “Three-dimensional dynamic rupture simulation with a high-order discontinuous Galerkin method on unstructured tetrahedral meshes”. In: Journal of Geophysical Research: Solid Earth 117.B2 (2012).

[78] Christian Pelties, A-A Gabriel, and J-P Ampuero. “Verification of an ADER-DG method for complex dynamic rupture problems”. In: Geoscientific Model Development 7.3 (2014), pp. 847–866.

[79] Yoshiaki Ida. “Cohesive force across the tip of a longitudinal-shear crack and Griffith’s specific surface energy”. In: Journal of Geophysical Research 77.20 (1972), pp. 3796–3805.

[80] Ruth A Harris et al. “The SCEC/USGS dynamic earthquake rupture code verification exercise”. In: Seismological Research Letters 80.1 (2009), pp. 119–126.

[81] Jeremy E Kozdon, Eric M Dunham, and Jan Nordström. “Simulation of dynamic earthquake ruptures in complex geometries using high-order finite difference methods”. In: Journal of Scientific Computing 55.1 (2013), pp. 92–124.

[82] Ronald L Biegel, Charles G Sammis, and Ares J Rosakis. “An experimental study of the effect of off-fault damage on the velocity of a slip pulse”. In: Journal of Geophysical Research: Solid Earth 113.B4 (2008).

[83] Yihe Huang and Jean-Paul Ampuero. “Pulse-like ruptures induced by low-velocity fault zones”. In: Journal of Geophysical Research: Solid Earth 116.B12 (2011).

[84] Yaron Finzi and Sebastian Langer. “Predicting rupture arrests, rupture jumps and cascading earthquakes”. In: Journal of Geophysical Research: Solid Earth 117.B12 (2012).

[85] Reza Abedi. “Spacetime damage-based cohesive model for elastodynamic fracture with dynamic contact”. PhD thesis. University of Illinois at Urbana-Champaign, 2010.

[86] Christian Howard. “Spacetime meshing of stratified spaces for spacetime discontinuous galerkin methods in arbitrary spatial dimensions”. MA thesis. University of Illinois at Urbana-Champaign, 2019.

Appendix A

Wrangler State Machines

We present the pseudo-code for the wrangle and store FSMs as described in Section 3.3.3.

A.1 Wrangle finite state-machine

Algorithm 18 Wrangle finite state-machine

INPUT: `v` - Base footprint vertex

STATE: `s` - Current state of wrangler; `holds_successful`: overall blocked status of footprint; `lock_successful`: overall lock status of footprint

```
procedure WRANGLE STATE MACHINE(v)
    switch s do
        case start :
            lock_successful ← true
            holds_successful ← true
            Request center vertex v
            If all requests complete: s ← getCenterVertex
        end case

        case getCenterVertex :
            Load center vertex v
            holds_successful ← holds_successful AND !v.is_blocked()
            for chambers inc_c incident to v do
                Request incident chamber inc_c
            end for
            If all requests completed: s ← getIncidentChambers
        end case
```

▷ continued

```
        case getIncidentChambers :
            Load incident chambers inc_c
            for vertices link_v of chamber inc_c do
                Request link vertices link_v        ▷ Link vertices are those other than base vertex v
            end for
            If all requests completed: s ← getLinkVertices
        end case

        case getLinkVertices :
            Load link vertices link_v
            holds_successful ← holds_successful AND !link_v.is_blocked()
            for chambers ext_c incident to link_v do
                Request extended chamber ext_c        ▷ Extended chambers are those that share an edge
                                                      ▷ with incident chambers inc_c
            end for
            If all requests completed: s ← getExtendedChambers
        end case

        case getExtendedChambers :
            Load extended chambers ext_c
            for vertices ext_v of chamber ext_c do
                Request extended vertices ext_v  ▷ Extended vertices ones not considered before
            end for
            If all requests completed: s ← getExtendedVertices
        end case

        case getExtendedVertices :
            Load extended vertices ext_v
            holds_successful ← holds_successful AND !ext_v.is_blocked()
            if holds_successful then                  ▷ All holds successfully acquired
                for all vertices fp_v in footprint do
                    Request to lock fp_v
                end for
                If all requests completed: s ← loadLockResults
            else
                lock_successful ← false
                s ← Complete
            end if
        end case
```
▷ continued

```
52:         case loadLockResults :
53:             Load lock results
54:             for all vertices fp_v  do
55:                 lock_successful ← lock_successful AND fp_v.successful_locked()
56:                 if fp_v.successful_locked() then      ▷ successfully acquired vertex lock on
    remote server
57:                     Add vertex to locked_vert_list
58:                 end if
59:             end for
60:             if lock_successful then
61:                 s ← Complete
62:                 lock_successful ← true
63:             else                ▷ Failed to acquire all locks, release ones that *were* acquired
64:                 for locked vertices l_v in locked_vert_list do
65:                     Request unlock for vertex l_v
66:                 end for
67:                 If all requests completed: s ← Complete
68:                 lock_successful ← false
69:             end if
70:         end case
71:     end switch
72: end procedure
```

A.2 Store finite state-machine

Algorithm 19 Store finite state-machine

INPUT: fp - Footprint to be stored
STATE: s - Current state of storer

```
 1: procedure Store State Machine(fp)
 2:     switch s do
 3:         case start :
 4:             s ← Erase
 5:         end case
```

▷ continued

```
 6:     case Erase :
 7:         for all chambers erase_c marked for erase in fp do
 8:             Request erase of erase_c
 9:         end for
10:         for all vertices erase_v marked for erase in fp do
11:             Request erase of erase_v
12:         end for
13:         s ← Insert
14:     end case

15:     case Insert :
16:         for all chambers ins_c marked for insert in fp do
17:             Determine server rank sr to perform insert, see Algorithm 13
18:             Request insert of ins_c on sr and await response for inserted global ID
19:         end for
20:         for all vertices ins_v marked for insert in fp do
21:             Determine server rank sr to perform insert, see Algorithm 13
22:             Request insert of ins_v on sr and await response for inserted global ID
23:         end for
24:         If all responses received: s ← UpdateIds
25:     end case

26:     case UpdateIds :
27:         Load ID responses
28:         for all chambers ins_c marked for insert in fp do
29:             Update local ID of ins_c with received global ID
30:             Mark ins_c for update
31:         end for
32:         for all vertices ins_v marked for insert in fp do
33:             Update local ID of ins_v with received global ID
34:             Mark ins_v for update
35:         end for
36:         s ← Update
37:     end case
38:     case Update :
39:         for all chambers upd_c marked for update in fp do
40:             Request update of upd_c
41:         end for
```
▷ continued

```
        for all vertices upd_v marked for update in fp do
            Request update of upd_v
            if upd_v a is local minima then
                Determine wrangler rank w to perform insert of minima, see Algorithm 14
                Request insert of minima on rank w
            end if
        end for
        s ← Unlock
    end case

    case Unlock :
        for all vertices unl_v marked for unlock in fp do
            Request unlock of unl_v
        end for
        s ← Complete
    end case
  end switch
end procedure
```