HIGH PERFORMANCE COMPUTING METHODS FOR EARTHQUAKE CYCLE SIMULATIONS

by

ALEXANDRE CHEN

A DISSERTATION

Presented to the Department of Computer Science
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2024

DISSERTATION ABSTRACT

Alexandre Chen

Doctor of Philosophy

Department of Computer Science

June 2024

Title: High Performance Computing Methods for Earthquake Cycle Simulations

Earthquakes often occur on complex faults of multiscale physical features, with different time scales between seismic slips and interseismic periods for multiple events. Single event, dynamic rupture simulations have been extensively studied to explore earthquake behaviors on complex faults, however, these simulations are limited by artificial prestress conditions and earthquake nucleations. Over the past decade, significant progress has been made in studying and modeling multiple cycles of earthquakes through collaborations in code comparison and verification. Numerical simulations for such earthquakes lead to large-scale linear systems that are difficult to solve using traditional methods in this field of study. These challenges include increased computation and memory demands. In addition, numerical stability for simulations over multiple earthquake cycles requires new numerical methods. Developments in High performance computing (HPC) provide tools to tackle some of these challenges. HPC is nothing new in geophysics since it has been applied in earthquake-related research including seismic imaging and dynamic rupture simulations for decades in both research and industry. However, there's little work in applying HPC to earthquake cycle modeling. This dissertation presents a novel approach to applying the latest advancements in HPC and numerical methods to solving computational challenges in earthquake cycle simulations.

CURRICULUM VITAE

NAME OF AUTHOR: Alexandre Chen

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Fudan University, Shanghai, China

DEGREES AWARDED:

Bachelor of Science, Physics, 2018, Fudan University

AREAS OF SPECIAL INTEREST:

Itereative Algorithms
Preconditioner
Numerical PDEs
SBP-SAT finite difference methods
High performance computing

PROFESSIONAL EXPERIENCE:

Teaching Assistant, Department of Computer Science, University of
2019S, 2019F, 2020S, 2020F, 2021W, 2022W, 2023W, 2023F
Researcher, Frontier Development Lab, June - August, 2022

GRANTS, AWARDS AND HONORS:

Graduate Teaching/Research Fellowship, University of Oregon

PUBLICATIONS:

Alexandre Chen, Brittany A. Erickson, Jeremy Kozdon, and Jee Choi. 2024. Matrix-free SBP-SAT finite difference methods and the multigrid precondi- tioner on GPUs. In Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 4–7, 2024, Kyoto, Japan. https://doi.org/10.1145/3650200.3656614

Brittany A. Erickson et. al. (2022) Incorporating Full Elastodynamic Effects and Dipping Fault Geometries in Community Code Verification Exercises for Simulations of Earthquake Sequences and Aseismic Slip (SEAS) Bulletin of the Seismological Society of America, https://doi.org/10.1785/0120220066

ACKNOWLEDGEMENTS

To do ...

This work benefited from access to the University of Oregon high performance computing cluster, Talapas, and Oregon Advanced Computing Institute for Science and Society (OACISS). This work is supported by NSF awards #2053372 and #2036980.

To so-and-so...

TABLE OF CONTENTS

Chapter Page

Chapter Page

Chapter Page

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

CHAPTER I

INTRODUCTION

1.1 Introduction to earthquakes

Every year, around 20,000 earthquakes happen around the world. Some are not noticeable, while others cause huge damage to property and life. Earthquakes have been recorded for thousands of years, and they are considered as signs or punishments to humans from supernatural powers in many civilizations. Over the past centuries, with advancements in mathematics, physics, geology, and other natural sciences, we have a more structured understanding of earthquakes today.

An earthquake represents a complex process of fault slip and energy release within the Earth's crust, driven by tectonic forces and resulting in the shaking of the ground and potentially causing damage to structures and infrastructure. An earthquake occurs due to the sudden release of accumulated stress along a fault line, resulting in rapid movement known as fault slip. This movement can be described in terms of several key components:

1.1.1 Fault rupture. The earthquake begins with the rupture of the fault, where the stress accumulated along the fault plane exceeds the strength of the rocks, causing them to fracture and slide past each other. This rupture initiates the seismic event.

1.1.2 Sesmic waves. As the fault ruptures, it generates seismic waves that propagate through the Earth's crust and propagate outward from the fault. These weaves transmit energy in the form of vibrations, which cause the ground to shake.

1.1.3 Slip motion. Slip motions are the noticeable components of an earthquake movement. The fault slip during an earthquake involves two types of movements

- Primary slip (seismic slip): This is the sudden and rapid movement along the fault plane during the initial rupture. It is usually associated with the intense shaking and damage from the earthquakes
- Afterslip: Following the primary slip, there may be additional movement along the fault. This ongoing slip can continue for days, weeks, or even months after the initial earthquake.

1.1.4 Displacement. The amount of fault slip during an earthquake is measured in terms of displacements. This is the distance that one side of the fault moves relative to the other. Displacement can be horizontal, vertical, or both.

1.2 Seismic and aseismic slip

The slip motion of an earthquake can further be classified into seismic and aseismic slips.

1.2.1 Seismic slip. Seismic slip refers to the sudden release of accumulated tectonic stress along a fault plane, resulting in what is often called an earthquake. This type of fault slip is characterized by rapid and dynamic movement, which generates seismic waves propagating through Earth's crust, causing ground shake and potential damages to infrastructures. Seismic fault slip occurs when stress accumulated along a fault exceeds the frictional resistance holding the fault surface, causing sudden slip and rupture.

1.2.2 Aseismic slip. Aseismic slip, also known as creep or slow slip, refers to gradual continuous movement along a fault plane without generating significant earthquakes and seismic waves. Unlike seismic slip, seismic slip occurs at rates that are usually much slower and may not produce noticeable ground shaking or seismic activity. Instead, seismic slip represents a steady release of tectonic stress along the fault, often occurring in between larger seismic events. However, the stress could still increase during seismic slip, leading to seismic slip in the future. Aseismic slip can contribute to the overall deformation of the Earth's crust and could play a potential role in seismic hazard assessments and forecasting. Modeling the behavior of aseismic slip has been essential in order to understand the nucleation of seismic slip and the mechanism behind multiple cycles of earthquakes.

1.2.3 Budget. In the context of seismology, budget refers to the distribution and allocation of accumulated tectonic stress or energy between seismic and aseismic slip events. Understanding the balance between seismic and aseismic slip budgets is important for assessing seismic hazard and fault behavior. It helps researchers and geoscientists to understand the mechanisms governing fault movement and stress release to forecast earthquakes. Here's a great review by Jean-Philippe discussing principles of fault slip budget determination and observational constraints on seismic and aseismic slip Avouac (2015).

1.3 Velocity weakening/strengthening

Understanding the friction behavior along the fault is the key to understanding the different behaviors of seismic and aseismic slips. Friction is often associated with the velocity of the fault displacement. Based on the different responses to sliding velocity, regions on the fault can be classified into two types: velocity weakening and velocity strengthening.

1.3.1 Veclocity weakening region. In a velocity weakening region, the friction resistance between the fault surfaces decreases with an increasing slip velocity. In other words, as the sliding velocity along the fault increases, the friction strength decreases.

This phenomenon is crucial in earthquake dynamics because it promotes the instability of the fault and facilitates the rapid release of accumulated stress during earthquakes. When the friction resistance decreases with increasing velocity, the fault becomes more prone to slip suddenly and can generate earthquake waves. Velocity weakening regions are often associated with materials or conditions that exhibit unstable slip behavior, such as fault gouge, pore pressure, and fluids.

1.3.2 Velocity strengthening region. In contrast to velocity weakening region, a velocity strengthening region is characterized by an increase in frictional resistance with increasing slip velocity. The velocity strengthening regions tend to promote stable fault behavior, resisting slip and preventing rapid release of stress that leads to earthquakes. Instead, fault slip in these regions may occur aseismically.

Velocity strengthening is typically observed in the shallow crust and at greater depths where ductile deformation mechanisms dominate. Conditions such as high confining pressure and high temperature, which promote stable creep and plastic deformation, contribute to this behavior.

1.4 Rate-and-state friction law

Since the recognition that earthquakes probably represent frictional slip instabilities in the 1960s, interest in determining frictional properties has been increased. The stability of frictional sliding depends on whether frictional resistance increases or drops during slip. Laboratory experiments have shown that frictional sliding is mainly a rate-dependent process in a steady-state regime with constant stress and steady velocity. Due to this, a state variable needs to be introduced to describe

- the transient behavior observed in non-steady-state experiments, denoted as a. This parameter presents the direct effect of the fault slip rate on the frictional resistance. In many rate-and-state friction laws, an increase in slip rate tends to increase the frictional resistance, and a quantifies the strength of this effect. A higher value of a means that the frictional resistance increases more rapidly with slip rate

- healing in hold-and-stick experiments denoted as b. This parameter represents the evolutionary effect of the fault slip on the frictional resistance. It quantifies how the frictional resistance evolves over time due to slip history. A positive value of b means that slip tends to decrease the frictional resistance over time, making fault slip easier in the future. A negative value of b implies the opposite, where slip tends to increase the frictional resistance over time, making fault slips harder.

These parameters are often determined empirically through laboratory experiments or field study. They play a crucial role in modeling the dynamics of earthquakes.

The formulation of such rate-and-state variable has significantly simplified the impacts of several parameters from rheology on the slip rate, which enables the development of numerical models to simulate earthquake cycles.

For most materials, it has been revealed by laboratory experiments on frictional sliding that the following conditions exist:

- The resistance to sliding depends on the sliding rate at steady rate, along with a logarithmic dependency of the coefficient of friction on the slip rate

- The resistance to sliding increases to a transient peak value when the imposed slip rate is suddenly changed, with the peak value being a logarithmic function of the slip rate

- The friction coefficient is approximately a linear function with the logarithmic of the time in hold-and-slip experiments.

Laboratory measurements at slow sliding rates can be reproduced relatively well with a rate-and-state formalism on the order of microns per second. Various laws have been proposed (Dieterich (1979a, 1979b); Marone (1998); Ruina (1983)).

One common such law is called the aging law

$$\mu = \mu_* + (a-b)\ln\frac{V}{V_*} \tag{1.1}$$

$$\frac{d\theta}{dt} = 1 - \frac{V\theta}{D_c} \tag{1.2}$$

At steady state, the law is purely rate dependent

$$\mu_{ss} = \mu_* + (a-b)\ln\frac{V}{V_*} \tag{1.3}$$

For a single-degree-of-freedom system such as a spring-and-slider system, the stability analysis shows that the slip can be stable only if $a - b > 0$ and that unstable slip requires that $a - b < 0$ For unstable slip to occur, it requires $a - b$ that is smaller than a critical negative value, defining an intermediate domain of conditional stability. For a crack with size L embedded in an elastic medium with shear modulus G, the condition for unstable slip is

$$a - b < -\lambda\frac{GD_c}{L\sigma_n'} \tag{1.4}$$

where λ is on the order of unity. σ_n' is the effective normal stress with $\sigma_n' - \sigma_n - P$ where P is pore pressure. In the limit when the pore pressure becomes near lithostatic, the critical value becomes infinite. This implies that high pore pressure should promote stable slip through the reduction of the effective normal stress. The rate-and-state friction and many definitions here are important concepts in the earthquake cycle simulations that are going to be discussed in later chapters.

1.5 SEAS project

The progress in understanding earthquake behaviors from rate-and-state friction laws makes numerical modeling of earthquakes possible. Many numerical models have been proposed to understand earthquake behaviors from single earthquakes to multiple earthquake simulations.

1.5.1 The limits of dynamic ruptures and earthquake simulators. For individual earthquakes, dynamic rupture simulations have been applied to study the influence of fault structure, geometry, constitutive laws, and prestress on earthquake rupture propagation and associated ground motion. These simulations are limited to single-event scenarios with limited timescales (seconds to minutes) and are affected by artificial prestress conditions and ad hoc nucleation procedures. The other approach uses earthquake simulators aiming to produce complex spatiotemporal characteristics of seismicity over millennial time scales, but these simulators simplify and approximate several key physical features that could influence or dominate earthquake and fault interactions to make these large-scale simulations computationally tractable

Tullis et al. (2012). The missing physical effects, such as seismic slip, wave-mediated dynamic stress transfers, and inelastic bulk response have the potential to dominate earthquake and fault interactions but are failed to be captured by these earthquake simulators

1.5.2 The importance of earthquake cycle simulations. A modeling framework to capture features and missing parts of the dynamic rupture simulations and earthquake simulators are simulations of sequences of earthquakes and aseismic slip (SEAS). These SEAS models focus on smaller, regional-scale fault zones and are designed to figure out physical factors that control the full range of observations of seismic slip, nucleation locations and actual earthquakes (dynamic rupture), ground shaking, etc. Such SEAS models can reveal initial conditions and earthquake nucleation for dynamic ruptures and identify important physical ingredients, as well as appropriate numerical approximations that could be later used in larger-scale, longer-term earthquake simulators.

Earlier methods for SEAS simulations have simplified assumptions including a linear elastic material response, approximate elastodynamic effects, and simple fault geometries in the 2D domain to ease computational demands. The first two benchmark problems proposed, BP1-QD and BP2-QD, use a relatively simple setup (2D anti-plane problem, with a vertically embedded, planar fault). They are designed to test the capabilities of different computational methods in correctly solving a mathematically well-defined basic problem. Good agreements across codes are obtained in terms of the number of characteristic events and recurrence times, as well as short-term processes (maximum slip rates, stress drops, and rupture speeds) when numerical parameters are chosen properly, especially when the computational domain is chosen large enough with sufficient resolution (small enough cell size). During these code tests, pure volume-based codes need to discretize a 2D domain and determine values for dimensions in 2D that are sufficiently large. Because of this, the exploration of computational domain size is an expensive task. To ease computations, grid stretching is applied to allow higher resolution around the fault or in the vicinity of the frictional portion of the fault. However, this does not propose a generic approach to tackle the computational challenge for these volume-based numerical methods, and the issue will be more challenging in simulations for 3D benchmark problems.

CHAPTER II

METHODOLOGY

2.1 Numerical methods

Computational modeling of the natural world involves pervasive material and geometric complexities that are hard to understand, incorporate, and analyze. The partial differential equations (PDEs) governing many of these systems are subject to boundary and interface conditions, and all numerical methods share the fundamental challenge of how to enforce these conditions in a stable manner. Additionally, applications involving elliptic PDEs or implicit time-stepping require efficient solution strategies for linear systems of equations.

Most applications in the natural sciences are characterized by multiscale features in both space and time which can lead to huge linear systems of equations after discretization. Our work is motivated by large-scale (∼hundreds of kilometers) earthquake cycle simulations where frictional faults are idealized as geometrically complex interfaces within a 3D material volume and are characterized by much smaller-scale features (∼microns) B. A. Erickson and Dunham (2014); Kozdon, Dunham, and Nordström (2012). In contrast to the single-event simulations, e.g. Roten et al. (2016), where the computational work at each time step is a single matrix-vector product, earthquake cycle simulations must integrate with adaptive time-steps through the slow periods between earthquakes, and are tasked with a much more costly linear solve. For example, even with upscaled parameters so that larger grid spacing can be used, the 2D simulations in B. A. Erickson and Dunham (2014) generated matrices of size $\sim 10^6$, and improved resolution and 3D domains would increase the system size to $\sim 10^9$ or greater. Because iterative schemes are most often implemented for the linear solve (since direct methods require a matrix factorization that is often too large to store in memory), it is no surprise that the sparse matrix-vector product (SpMV) arises as the main computational workhorse. The matrix sparsity and condition number depend on several physical and numerical factors including the material heterogeneity of the Earth's material properties, order of accuracy, the coordinate transformation (for irregular grids), and the mesh size. For large-scale problems, matrix-free (on-the-fly) techniques for the SpMV are fundamental when the matrix cannot be stored explicitly.

In this work, we use summation-by-parts (SBP) finite difference methods Kreiss and Scherer (1974); Mattsson and Nordström (2004); Strand (1994); Svärd and Nordström (2014), which are distinct from traditional finite difference methods in their use of specific one-sided approximations at domain boundaries that enable the highly valuable proof of stability, a necessity for numerical convergence. Weak enforcement of boundary conditions has additional superior properties over traditional methods, for example, the simultaneous-approximation-term (SAT) technique, which relaxes continuity requirements (of the grid and the solution) across physical or geometrical interfaces, with low communication overhead for efficient parallel algorithms Del Rey Fernández, Hicken, and Zingg (2014).

For these reasons SBP-SAT methods are widely used in many areas of scientific computing, from the flow over airplane wings to biological membranes to earthquakes and tsunamigenesis B. A. Erickson and Day (2016); Lotto and Dunham (2015); Nordström and Eriksson (2010); Petersson and Sjögreen (2012); Swim et al. (2011); Ying and Henriquez (2007); these studies, however, have not been developed for linear solves or were limited to small-scale simulations.

With this chapter, we review the SBP-SAT methods and how they are used to formulate linear systems. We then review several numerical methods used to solve the linear system including multigrid methods. We contribute a novel iterative scheme for linear systems based on SBP-SAT discretizations where nontrivial computations arise due to boundary treatment. These methods are integrated into our existing, public software for simulations of earthquake sequences. Specifically, we make the following contribution in preconditioning specifically:

Since preconditioning of iterative methods is a hugely consequential step towards improving convergence rates, we develop a custom geometric multigrid preconditioned conjugate gradient (MGCG) algorithm which shows a near-constant number of iterations with increasing system size. The required iterations (and time-to-solution) are much lower compared to several off-the-shelf preconditioners offered by the PETSc library Balay et al. (2023), a state-of-the-art library for scientific computing.

Furthermore, the ubiquity of SBP-SAT methods in modern scientific computing applications means our work has the propensity to advance scientific studies currently limited to small-scale problems.

2.2 SBP-SAT methods

SBP methods approximate partial derivatives using one-sided differences at all points close to the boundary node, generating a matrix approximating a partial derivative operator. In this work we focus on second-order derivatives which appear in (4.3), however the matrix-free methods we derive are applicable to any second-order PDE. We consider SBP finite-difference approximations to boundary-value problem (4.3), i.e. on the square computational domain $\bar{\Omega}$; solutions on the physical domain Ω are obtained by the inverse coordinate transformation.

In this work, we focus on SBP operators with second-order accuracy which contains abundant complexity at domain boundaries to enable insight into implementation design extendable to higher-order methods. To provide background on the SBP methods we first describe the 1D operators, as Kronecker products are used to form their multi-dimensional counterparts.

2.2.0.1 1D Operators. We discretize the spatial domain $-1 \leq r \leq 1$ with $N+1$ evenly spaced grid points $r_i = -1 + ih, i = 0, \ldots, N$ with grid spacing $h = 2/N$. A function u projected onto the computational grid is denoted by $\boldsymbol{u} = [u_0, u_1, \ldots, u_N]^T$ and is often taken to be the interpolant of u at the grid points. We define the grid basis vector $\vec{e}_j$ to be a vector with value 1 at grid point j and 0 for the rest, which allows us to extract the jth component: $u_j = \vec{e}_j^T \vec{u}$.

Definition 1 (First Derivative)**.** *A matrix $\boldsymbol{D}_r$ is an SBP approximation to the first derivative operator $\partial/\partial r$ if it can be decomposed as $\boldsymbol{H}\boldsymbol{D}_r = \boldsymbol{Q}$ with $\boldsymbol{H}$ being SPD and $\boldsymbol{Q}$ satisfying $\vec{u}^T(\boldsymbol{Q} + \boldsymbol{Q}^T)\vec{v} = u_N v_N - u_0 v_0$.*

Here, $\boldsymbol{H}$ is a diagonal quadrature matrix and $\boldsymbol{D}_r$ is the standard central finite difference operator in the interior which transitions to one-sided at boundaries. The reason why the operator D_x is called SBP is that it mimics the integration-by-part property

$$\int_0^1 u\frac{\partial v}{\partial x} + \int_0^1 \frac{\partial u}{\partial x} v = uv\Big|_0^1, \tag{2.1}$$

in a discrete form

$$\vec{u}^T \boldsymbol{H}\boldsymbol{D}_x\vec{v} + \vec{u}^T \boldsymbol{D}_x^T\boldsymbol{H}\vec{v} = \vec{u}^T\left(\boldsymbol{Q} + \boldsymbol{Q}^T\right)\vec{v} = u_N v_N - u_0 v_0. \tag{2.2}$$

Definition 2 (Second Derivative)**.** *Letting $c = c(r)$ denote a material coefficient, we define matrix $\boldsymbol{D}_{rr}^{(c)}$ to be an SBP approximation to $\frac{\partial}{\partial r}\left(c\frac{\partial}{\partial r}\right)$ if it can be decomposed as $\boldsymbol{D}_{rr}^{(c)} = \boldsymbol{H}^{-1}(-\boldsymbol{M}^{(c)} +$*

$c_N \vec{e}_N \vec{d}_N^T - c_0 \vec{e}_0 \vec{d}_0^T$) *where* $\boldsymbol{M}^{(c)}$ *is SPD and* $\vec{d}_0^T \vec{u}$ *and* $\vec{d}_N^T \vec{u}$ *are approximations of the first derivative of* u *at the boundaries.*

Similarly, the operator $\boldsymbol{D}_{xx}^{(c)}$ mimics the integration-by-parts property

$$\int_0^1 u \frac{\partial}{\partial x}\left(c \frac{\partial v}{\partial x}\right) + \int_0^1 \frac{\partial u}{\partial x} c \frac{\partial v}{\partial x} = uc \frac{\partial v}{\partial x}\bigg|_0^1, \tag{2.3}$$

in a discrete form

$$\vec{u}^T \boldsymbol{H} \boldsymbol{D}_{xx}^{(c)} \vec{v} + \vec{u}^T \boldsymbol{A}^{(c)} \vec{v} = c_N u_N \vec{d}_N^T \vec{v} - c_0 u_0 \vec{d}_0^T \vec{v}. \tag{2.4}$$

With these properties, both $\boldsymbol{D}_r$ and $\boldsymbol{D}_{rr}^{(c)}$ mimic integration-by-parts in a discrete form which enables the proof of discrete stability Mattsson, Ham, and Iaccarino (2009); Mattsson and Nordström (2004).

$\boldsymbol{D}_{rr}^{(c)}$ is a centered difference approximation within the interior of the domain, but includes approximations at boundary points as well. For illustrative purposes alone, if $c = 1$ (e. g. a constant coefficient case), the matrix is given by

$$\boldsymbol{D}_{rr}^{(c)} = \frac{1}{h^2} \begin{bmatrix} 1 & -2 & 1 & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & 1 & -2 & 1 & \\ & & 1 & -2 & 1 & \end{bmatrix},$$

which, as highlighted in red, resembles the traditional (second-order-accurate) Laplacian operator in the domain interior.

2.2.0.2 2D SBP Operators. The 2D domain $\bar{\Omega}$ is discretized using $N+1$ grid points in each direction, resulting in an $(N+1) \times (N+1)$ grid of points where grid point (i, j) is at $(x_i, y_j) = (-1 + ih, -1 + jh)$ for $0 \leq i, j \leq N$ with $h = 2/N$. Here we have assumed equal grid spacing in each direction, only for notational ease; the generalization to different numbers of grid points in each dimension does not impact the construction of the method and is implemented in our code. A 2D grid function $\boldsymbol{u}$ is ordered lexicographically and we let $\boldsymbol{C}_{ij} = \text{diag}(\boldsymbol{c}_{ij})$ define the diagonal matrix of coefficients, see Kozdon, Erickson, and Wilcox (2020).

In this work we imply summation notation whenever indices are repeated. Multi-dimensional SBP operators are obtained by applying the Kronecker product to 1D operators,

for example, the 2D second derivative operators are given by

$$\begin{aligned}\frac{\partial}{\partial i} c_{ij} \frac{\partial}{\partial j} &\approx \tilde{\boldsymbol{D}}_{ij}^{c_{ij}} \\ &= (\boldsymbol{H} \otimes \boldsymbol{H})^{-1} \left[-\tilde{\boldsymbol{M}}_{ij}^{(c_{ij})} + \boldsymbol{T} \right], \end{aligned} \tag{2.5}$$

for $i, j \in \{r, s\}$. Here $\tilde{\boldsymbol{M}}_{ij}^{(c_{ij})}$ is the sum of SPD matrices approximating integrated second derivatives (i.e. sum over repeated indices i, j) for example $\int_{\bar{\Omega}} \frac{\partial}{\partial r} c_{rr} \frac{\partial}{\partial r} \approx \tilde{\boldsymbol{M}}_{rr}^{(c_{rr})}$ and matrix $\boldsymbol{T}$ involves the boundary derivative computations, see B. A. Erickson, Kozdon, and Harvey (2022) for complete details.

2.2.0.3 SAT Penalty Terms. SBP methods are designed to work with various impositions of boundary conditions that lead to provably stable methods, for example through weak enforcement via the simultaneous-approximation-term (SAT) Carpenter, Gottlieb, and Abarbanel (1994) which we adopt here. As opposed to traditional finite difference methods that "inject" boundary data by overwriting grid points with the given data, the SAT technique imposes boundary conditions weakly (through penalization), so that all grid points approximate both the PDE and the boundary conditions up to a certain level of accuracy. The combined approach is known as SBP-SAT. Where traditional methods that use injection or strong enforcement of boundary/interface conditions destroy the discrete integration-by-parts property, using SAT terms enables proof of the method's stability (a necessary property for numerical convergence) Mattsson (2003).

2.2.1 An example of the SBP-SAT technique for PDE.

We use the following example from Ruggiu, Weinerfelt, and Nordström (2018) to showcase an example of applying the SBP-SAT method for PDEs. Let's consider the advection problem in 1D.

$$\begin{aligned} \boldsymbol{u}_t + \boldsymbol{u}_x &= 0, \quad 0 < x < 1, t > 0 \\ \boldsymbol{u}(0, t) &= \boldsymbol{g}(t), \quad t > 0 \\ \boldsymbol{u}(x, t) &= \boldsymbol{h}(x), \quad 0 < x < 1 \end{aligned} \tag{2.6}$$

where both g and h are known for initial and boundary conditions. The problem Equation 2.6 has an energy-estimate and is well-posed. We can easily learn that the analytical solution for this equation is a right-traveling wave.

We discretize 1D domain with $N+1$ points in a uniform grid on $[0,1]$ using the method described in **??**. By applying SBP-SAT discretization in space to Equation 2.6, we get

$$\begin{aligned} \boldsymbol{u}_t + D_1\boldsymbol{u} &= P^{-1}\sigma(\boldsymbol{u}_0 - \boldsymbol{g})\boldsymbol{e}_0, \quad t > 0 \\ \boldsymbol{u}(0) &= \boldsymbol{h} \end{aligned} \tag{2.7}$$

where $\boldsymbol{u} = [u_0, \ldots, u_N]^T, \boldsymbol{h} = [h_0, \ldots, h_N]^T, \sigma \in \mathbb{R}$ is a penalty parameter which is determined through stability condition. $\boldsymbol{e}_0 = [1, 0, \ldots, 0]^T \in \mathbb{R}^{N+1}$. To determine the value for σ so that the problem Equation 2.7 is strongly stable, we have

$$||\boldsymbol{u}(t)||^2 \leq K(t)(||\boldsymbol{h}||^2 + \max_{\tau\in[0,t]}|\boldsymbol{g}(\tau)|^2) \tag{2.8}$$

The $K(t)$ in Equation 2.8 is independent of the data and bounded for any finite t and meshsize Δx. Further details about $K(t)$ are given in Gustafsson, Kreiss, and Oliger (1995); Svärd and Nordström (2014). Applying the energy method by multiplying the equation Equation 2.7 with $\boldsymbol{u}^T P$ and adding the transpose with the SBP property Equation 2.2, we find

$$\frac{d}{dt}||\boldsymbol{u}||_P^2 = -\frac{\sigma^2}{1+2\sigma}\boldsymbol{g}^2 - \boldsymbol{u}_N^2 + \frac{[(1+2\sigma)\boldsymbol{u}_0 - \sigma\boldsymbol{g}]^2}{1+2\sigma} \tag{2.9}$$

By time-integration, this leads to an estimate of the form Equation 2.8 for $\sigma < -1/2$.

2.2.2 Poisson's equation with SBP-SAT Methods. We consider the 2D Poisson equation on the unit square Ω with both Dirichlet and Neumann conditions for generality, as each appears in earthquake problems (e.g. Earth's free surface manifests as a Neumann condition, and the slow motion of tectonic plates is usually enforced via a Dirichlet condition). This is an important and necessary first step before additional complexities such as variable material properties, complex geometries, and fully 3D problems. The governing equations are given by

$$-\Delta u = f, \qquad \text{for } (x,y) \in \Omega, \tag{2.10a}$$

$$u = g_{\mathrm{W}}, \quad x = 0, \tag{2.10b}$$

$$u = g_{\mathrm{E}}, \quad x = 1, \tag{2.10c}$$

$$\boldsymbol{n}\cdot\nabla u = g_{\mathrm{S}}, \quad y = 0, \tag{2.10d}$$

$$\boldsymbol{n}\cdot\nabla u = g_{\mathrm{N}}, \quad y = 1, \tag{2.10e}$$

where $\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$, the field $u(x, y)$ is the unknown particle displacement, the scalar function $f(x, y)$ is the source function, and vector $\boldsymbol{n}$ is the outward pointing normal to the domain boundary $\partial\Omega$. The g's represent boundary data on the west, east, south, and north boundaries.

The SBP-SAT discretization of (2.10) is given by

$$-\boldsymbol{D}_2\boldsymbol{u} = \boldsymbol{f} + \boldsymbol{b}^N + \boldsymbol{b}^S + \boldsymbol{b}^W + \boldsymbol{b}^E, \tag{2.11}$$

where $\boldsymbol{D}_2 = (\boldsymbol{I} \otimes \boldsymbol{D}_{xx}) + (\boldsymbol{D}_{yy} \otimes \boldsymbol{I})$ is the discrete Laplacian operator and $\boldsymbol{u}$ is the grid function approximating the solution, formed as a stacked vector of vectors. The SAT terms $\boldsymbol{b}^N, \boldsymbol{b}^S, \boldsymbol{b}^W, \boldsymbol{b}^E$ enforce all boundary conditions weakly. To illustrate the structure of these vectors, the SAT term enforcing Dirichlet data on the west boundary is given by

$$\boldsymbol{b}^W = \alpha \left(\boldsymbol{H}^{-1} \otimes \boldsymbol{I}\right) \left(\boldsymbol{E}_{\mathrm{W}}\boldsymbol{u} - \boldsymbol{e}_{\mathrm{W}}^T\boldsymbol{g}_{\mathrm{W}}\right) \tag{2.12}$$

$$- \left(\boldsymbol{H}^{-1}\boldsymbol{e}_{\boldsymbol{0}}\boldsymbol{d}_{\boldsymbol{0}}^T \otimes \boldsymbol{I}\right) \left(\boldsymbol{E}_{\mathrm{W}}\boldsymbol{u} - \boldsymbol{e}_{\mathrm{W}}^T\boldsymbol{g}_{\mathrm{W}}\right), \tag{2.13}$$

where α again represents a penalty parameter, $\boldsymbol{E}_{\mathrm{W}}$ is a sparse boundary extraction operator, and $\boldsymbol{e}_{\mathrm{W}}^T$ is an operator that lifts the boundary data to the whole domain. Details of all the SAT terms can be found in B. A. Erickson and Dunham (2014). System (4.10) can be rendered SPD by multiplying from the left by $(\boldsymbol{H} \otimes \boldsymbol{H})$, producing the sparse linear system $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}$.

2.3 Iterative Methods

This section will present a brief review of iterative methods for solving large linear systems in our research. Many concepts and theorems are presented in Saad (2003) and we will point the detailed information and proofs to the book.

2.3.1 Stationary Iterative Methods.

Stationary iterative methods can be expressed in the simple form

$$\boldsymbol{u}^{k+1} = \boldsymbol{Q}\boldsymbol{u}^k + \boldsymbol{q} \tag{2.14}$$

where $\boldsymbol{Q}$ and $\boldsymbol{q}$ are placeholders for a matrix and a vector respectively, both independent of iteration step k. Stationary iterative methods, such as the Gauss-Seidel method, act as smoothers for damping high-frequency components of the solution vectors. Further backgrounds of these iterative methods can be found in Saad (2003)

The considered problem for iterative methods is a linear equation system of the form $\boldsymbol{Au} = \boldsymbol{f}$. We introduce splitting matrix $\boldsymbol{S}$ as follows:

$$\boldsymbol{A} = \boldsymbol{S} + (\boldsymbol{A} - \boldsymbol{S}) \tag{2.15}$$

With this introduced splitting matrix $\boldsymbol{S}$, we can rewrite the linear equation system as

$$\boldsymbol{Su} = (\boldsymbol{S} - \boldsymbol{A})\boldsymbol{u} + \boldsymbol{f} \tag{2.16}$$

and the iterative scheme of the splitting method is defined as

$$\boldsymbol{u}^{k+1} = \boldsymbol{S}^{-1}((\boldsymbol{S} - \boldsymbol{A})\boldsymbol{u}^k + f) \tag{2.17}$$

For further analysis, it is useful to introduce the iteration matrix $\boldsymbol{M}$ as

$$\boldsymbol{M} = \boldsymbol{S}^{-1}(\boldsymbol{S} - \boldsymbol{A}) \tag{2.18}$$

.

If we define $\boldsymbol{Q} = \boldsymbol{M}$ and $\boldsymbol{q} = \boldsymbol{S}^{-1}\boldsymbol{f}$, then Equation 2.17 can be written as $\boldsymbol{u} = \boldsymbol{Qu} + \boldsymbol{q}$, and it satisfies

$$\boldsymbol{e}^{k+1} = \boldsymbol{M}\boldsymbol{e}^k \tag{2.19}$$

where $\boldsymbol{e}^k$ is the error $\boldsymbol{u^k} - \boldsymbol{u}$ for the iteration step k. Because $\boldsymbol{M}$ is only determined by the initial linear system and the splitting matrix $\boldsymbol{S}$, and it is not changed in each iteration step, this method is called the stationary iterative method.

The spectral radius ρ is defined as the largest absolute eigenvalue of a matrix. The stationary iterative method converges if and only if the spectral radius ρ of the iteration matrix $\boldsymbol{M}$ satisfies the following condition

$$\rho(\boldsymbol{M}) < 1 \tag{2.20}$$

Such convergence holds for any initial guess $\boldsymbol{u}^0$ and any right-hand side $\boldsymbol{f}$. Different stationary iterative methods differ in the choice of splitting matrix $\boldsymbol{S}$. We will present the Jacobi method and the Gauss-Seidel method for comparison here.

2.3.1.1 The Jacobi Method. In the Jacobi method, the diagonal $\boldsymbol{D}$ of the matrix $\boldsymbol{A}$ for the linear system is chosen as the splitting matrix $\boldsymbol{S}$. Hence the decomposition is expressed

as

$$\boldsymbol{A} = \boldsymbol{D} + (\boldsymbol{A} - \boldsymbol{D}) \text{ or } \boldsymbol{A} = \boldsymbol{D} + (-\boldsymbol{L} - \boldsymbol{U}) \tag{2.21}$$

Where $-\boldsymbol{L}$ denotes the strictly lower triangle and $-\boldsymbol{U}$ denotes the strictly upper triangle of the matrix $\boldsymbol{A}$. Similar to Equation 2.17, the Jacobi method is then

$$\boldsymbol{u}^{k+1} = \boldsymbol{D}^{-1}((\boldsymbol{L} + \boldsymbol{U})\boldsymbol{u}^k + \boldsymbol{f}) \tag{2.22}$$

In terms of matrix indices, the Jacobi method can be written as

$$u_i^{k+1} = \frac{1}{A_{ii}}(f_i - \sum_{j=1, i \neq j} A_{ij} u_j^k) \tag{2.23}$$

For matrix-free forms, similar results can be obtained via slight modifications to this form.

2.3.1.2 The Gauss-Seidel Method. In the Gauss-Seidel method, the splitting matrix is chosen as $\boldsymbol{S} = (\boldsymbol{D} - \boldsymbol{L})$. The decomposition is then expressed as

$$\boldsymbol{A} = (\boldsymbol{D} - \boldsymbol{L}) + (\boldsymbol{A} - (\boldsymbol{D} - \boldsymbol{L})) \text{ or } \boldsymbol{A} = \boldsymbol{D} - \boldsymbol{L} + (-\boldsymbol{U}) \tag{2.24}$$

Similar to Equation 2.17, the Gauss-Seidel method is then

$$\boldsymbol{u}^{k+1} = (\boldsymbol{D} - \boldsymbol{L})^{-1}(\boldsymbol{U}\boldsymbol{u}^k + \boldsymbol{f}) \tag{2.25}$$

To derive the index form of the Gauss-Seidel method, some further transformations are needed.

$$\boldsymbol{D}\boldsymbol{u}^{k+1} - \boldsymbol{L}\boldsymbol{u}^{k+1} = \boldsymbol{U}\boldsymbol{u}^k + \boldsymbol{f} \tag{2.26}$$

and

$$\boldsymbol{u}^{k+1} = \boldsymbol{D}^{-1}(\boldsymbol{L}\boldsymbol{u}^{k+1} + \boldsymbol{U}\boldsymbol{u}^k + \boldsymbol{f}) \tag{2.27}$$

and the index form is given as

$$u_i^{k+1} = \frac{1}{A_{ii}}(f_i - \sum_{j=1}^{i-1} A_{ij} u_j^{k+1} - \sum_{i+1}^{n} A_{ij} u_j^k) \tag{2.28}$$

2.3.1.3 Comparison of the Jacobi and the Gauss-Seidel Method. It might seem that in Equation 2.27 the right-hand side contains the result from the iteration step k+1 and such an iterative scheme would fail. However, a closer observation would notice that the $\boldsymbol{u}^{k+1}$ is multiplied by the negation of the lower triangle $-\boldsymbol{L}$ of the matrix $\boldsymbol{A}$. This means for each

element j in the vector $\boldsymbol{u}^{k+1}$, only newly updated elements before index j are used, hence there is no logical problem in this iterative scheme. This is more obvious in the index form Equation 2.28

This is the most important difference between the Jacobi and the Gauss-Seidel method. When computing the i-th element u_i^{k+1} in the iteration step k+1, the Gauss-Seidel method already uses all available iterates u_j^{k+1} with $j = 1 \dots (i-1)$, while the Jacobi method only uses the iterates from the previous iteration step k. In other words, while the Jacobi method adds all increments *simultaneously* only after cycling through all degrees of freedom, the Gauss-Seidel method adds all increments *successively* as soon as available. As a result, the Gauss-Seidel method has the advantage that one vector is sufficient to update its vector elements i successively, in contrast to the Jacobi method where an additional vector is required. However, in terms of computational cost, the difference in memory requirement is negligible in practice.

On the other hand, the operations for the iteration of different vector elements do not coincide in the Jacobi method, which means the parallelization for the Jacobi method is straightforward. However, in the Gauss-Seidel method, the nodal ordering influences the convergence behavior. There are various nodal orderings summarized in Hackbusch (2013b), such as red-black, lexicographical, zebra-line, and four-color ordering. More advanced algorithms are required for the successful parallelization of the Gauss-Seidel method. Otherwise, uncontrolled splitting of the process leads to the so-called *chaotic* Gauss-Seidel method.

In terms of convergence, both stationary methods depend on the spectral radius of the corresponding iteration matrix $\boldsymbol{Q}$, which is affected by the splitting matrix $\boldsymbol{S}$ chosen for each of these two methods. If both methods converge, the convergence rate of the Gauss-Seidel method is better as each iteration would use the updated data as soon as available.

Specifically, it is sufficient for the Jacobi method to converge if the system Matrix $\boldsymbol{A}$ is strictly diagonally dominant Grossmann (1994)

$$|A_{ii}| > \sum_{j=1, j\neq i}^{n} |A_{ij}| \text{ for all } i \tag{2.29}$$

For a linear system that doesn't satisfy this condition, convergence can be achieved by additional damping. For the Gauss-Seidel method, other than the given condition in Equation 2.29, the convergence is also guaranteed if the system matrix A is positive definite. The second condition is usually satisfied for the finite difference method or the finite element method if properly restrained and stabilized. Bathe (2006) Other than directly used as standalone iterative

solvers, the damped Jacobi method or the Gauss-Seidel method can be applied as smoothers within the multigrid method.

2.3.1.4 Relaxation methods. Relaxation methods are also stationary iterative methods, thus they can also be presented in the form Equation 2.14. For each of the methods introduced previously, there also exists a corresponding relaxation method. In comparison to the precedent methods, the relaxation methods scale each increment by a constant relaxation factor ω. The general form of the relaxation methods is given by the following simplified algorithmic expression

$$u_i^{k+1} := (1-\omega)u_i^k + \omega \breve{u}_i^{k+1} \tag{2.30}$$

where for each individual index i, the temporary variable $\breve{u}_i^{k+1}$ is computed as the u_i^{k+1} of the Jacobi method in the case of the simultaneous over-relaxation method (JOR method) or as the u_i^{k+1} of the Gauss-Seidel method in the case of the successive over-relaxation method (SOR method). Thus when $\omega = 1$, the relaxation scheme is identical to the Jacobi or the Gauss-Seidel method.

The optimum relaxation factor can be derived theoretically from the spectral radius of the iteration matrix. However, this is expensive. For more practical use, several methods for the determination of ω were proposed in Grossmann (1994) and Young (2014).

2.3.2 Krylov Subspace Methods. Stationary iterative methods have been applied for a long time in history, but over the last few decades, Krylov subspace methods become more popular. These methods focus on building Krylov subspaces, named after Aleksei Nikolaevich Krylov who used these spaces to analyze oscillations of mechanical systems Krylov (1931). The Krylov subspace takes the form

$$\mathcal{K}_k(A, \boldsymbol{v}) := \text{span}\{\boldsymbol{v}, A\boldsymbol{v}, \ldots, A^{k-1}\boldsymbol{v}\} \tag{2.31}$$

where $A \in \mathbb{C}^{n\times n}$ and $\boldsymbol{v} \in \mathbb{C}^n$

2.3.2.1 Conjugate Gradient Method. The conjugate gradient (CG) method was developed by *Hestenes & Stiefel* Hestenes, Stiefel, et al. (1952) as one of the first Krylov subspace methods, and has been one of the most popular iterative methods in solving linear systems. Compared to the iterative methods mentioned in previous sections which are known to be stationary, the CG method is non-stationary. Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ be a symmetric positive definite

(SPD) matrix, and $\boldsymbol{f} \in \mathbb{R}^n$ be a real vector, then the minimization problem of the quadratic form $F(x) = min$

$$F(\boldsymbol{u}) = \frac{1}{2}\boldsymbol{u}^T\boldsymbol{A}\boldsymbol{u} - \boldsymbol{f}^T\boldsymbol{u} \tag{2.32}$$

is equivalent to getting its derivative

$$\text{grad}F(\boldsymbol{u}) = \boldsymbol{A}\boldsymbol{u} - \boldsymbol{f} \tag{2.33}$$

equal to the zero vector

$$\text{grad}F(\boldsymbol{u}) = 0 \tag{2.34}$$

The CG method is an iterative minimizer of the given quadratic form and therefore an iterative solver for the linear equation system $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{f}$ when $\boldsymbol{A}$ is SPD. The quadratic form is always minimized from an approximate vector $\boldsymbol{u}^k$ in the direction of a provided search vector $\boldsymbol{p}^k \neq 0$, which can be written as

$$F(\boldsymbol{u}^k + \lambda\boldsymbol{p}^k) = min \tag{2.35}$$

where both $\boldsymbol{u}^k$ and $\boldsymbol{p}^k$ are constant vectors $\in \mathbb{R}$ and a scalar variable $\lambda \in \mathbb{R}$. This leads to the following parabola function of λ

$$\begin{aligned}(\frac{1}{2}\boldsymbol{p}^{k^T}\boldsymbol{A}\boldsymbol{p}^k)\lambda^2 + (\boldsymbol{p}^{k^T}\boldsymbol{A}\boldsymbol{u}^k - \boldsymbol{p}^{k^T}\boldsymbol{f})\lambda \\ + ((\frac{1}{2}\boldsymbol{u}^{k^T}\boldsymbol{A}\boldsymbol{u}^k) - \boldsymbol{u}^{k^T}) = min\end{aligned} \tag{2.36}$$

This quadratic form is minimized for

$$\lambda = \frac{\boldsymbol{p}^{k^T}(\boldsymbol{f} - \boldsymbol{A}\boldsymbol{u}^k)}{\boldsymbol{p}^{k^T}\boldsymbol{A}\boldsymbol{p}^k} \tag{2.37}$$

The ideal search direction $\boldsymbol{p}^k$ would be the error $\boldsymbol{e}$, however, this would require us to know the exact solution $\boldsymbol{u}$. As a compromise, the negative gradient of the quadratic form at $\boldsymbol{u}^k$ is the best intuitive search direction from the local view of $\boldsymbol{u}^k$. The search direction corresponds to the residual $\boldsymbol{r}^k$ is now

$$-\text{grad}F(u^k) = \boldsymbol{f} - \boldsymbol{A}\boldsymbol{u}^k = \boldsymbol{r}^k \tag{2.38}$$

with $\boldsymbol{p}^k = \boldsymbol{r}^k$. We define the following equations

$$\lambda_k = \frac{\boldsymbol{r}^{k^T}\boldsymbol{r}^k}{\boldsymbol{r}^{k^T}\boldsymbol{A}\boldsymbol{r}^k} \tag{2.39}$$

$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \lambda_k\boldsymbol{r}^k \tag{2.40}$$

to describe the iterative process for one iterative step, which is called the method of steepest descent due to the fact that for any iteration step k, the search direction $\boldsymbol{p}^k$ is defined by (-grad$F(\boldsymbol{u}^k)$).

The method of the steepest descent is a key step in the CG method, but the choice of search directions $\boldsymbol{p}^k$ is not the optimal one. As $\boldsymbol{u}^{k+1}$ is optimized with respect to the previous search direction $\boldsymbol{p}^k = \boldsymbol{r}^k$, it is clear that the successive search directions are not orthogonal ($-\text{grad}F(\boldsymbol{u}^{k+1}) \perp \boldsymbol{p}^k$). It can be shown that $\boldsymbol{r}^k \perp \boldsymbol{r}^{k+1}$ and $\boldsymbol{r}^{k+1} \perp \boldsymbol{r}^{k+2}$, but it is general not true for $\boldsymbol{r}^k \perp \boldsymbol{r}^{k+2}$. Therefore, $\boldsymbol{u}^{k+1}$ has lost its optimum with respect to the previously optimized direction $\boldsymbol{r}^k$.

If $\boldsymbol{u}^{k+1}$ is optimal with respect to $\boldsymbol{p}^k \neq 0$, then this property is passed to $\boldsymbol{u}^{k+1}$ if and only if

$$A\boldsymbol{p}^{k+1} \perp \boldsymbol{p}^k \tag{2.41}$$

The vectors $\boldsymbol{p}^{k+1}$ and $\boldsymbol{p}^k$ are called *conjugate*. In the conjugate gradient method, the search directions are pairwise conjugate. Each time a new search direction is derived from the actual residual and conjugated with the prior search direction. It is also conjugate to all previous search directions. Thus a system of conjugate search directions is obtained or equivalent to a system of orthogonal residuals. This can be proven by induction. The initial values are defined as

$$\begin{aligned} \boldsymbol{r}^0 &= \boldsymbol{f} - \boldsymbol{A}\boldsymbol{u}^0 \\ \boldsymbol{p}^0 &= \boldsymbol{r}^0 \end{aligned} \tag{2.42}$$

The following equations describe the algorithm of the conjugate gradient method

$$\lambda_k = \frac{{\boldsymbol{r}^k}^T \boldsymbol{p}^k}{{\boldsymbol{p}^k}^T \boldsymbol{A}\boldsymbol{p}^k} \tag{2.43}$$

$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \lambda_k \boldsymbol{p}^k \tag{2.44}$$

$$\boldsymbol{r}^{k+1} = \boldsymbol{r}^k - \lambda_k \boldsymbol{A}\boldsymbol{p}^k \tag{2.45}$$

$$\boldsymbol{p}^{k+1} = \boldsymbol{r}^k - \frac{{\boldsymbol{r}^{k+1}}^T \boldsymbol{A}\boldsymbol{p}^k}{{\boldsymbol{p}^k}^T \boldsymbol{A}\boldsymbol{p}^k} \boldsymbol{p}^k \tag{2.46}$$

As shown in Grossmann (1994), for an efficient implementation, it is possible to use an alternative form for λ_k and p^{k+1}

$$\lambda_k = \frac{\boldsymbol{r}^{k^T}\boldsymbol{r}^k}{\boldsymbol{p}^{k^T}\boldsymbol{A}\boldsymbol{p}^k} \tag{2.47}$$

$$\boldsymbol{p}^{k+1} = \boldsymbol{r}^k + \frac{\boldsymbol{r}^{k+1^T}\boldsymbol{r}^k}{\boldsymbol{r}^{k^T}\boldsymbol{r}^k}\boldsymbol{p}^k \tag{2.48}$$

It can be proven that the CG method will converge to the exact solution after given finite steps. In theory, this method can achieve the same level of accuracy as a direct solver. However, due to numerical round-off errors, the orthogonality is often lost and such ideal theoretical results can not be achieved. In practice, given reasonable error tolerance, the CG method can generally be terminated after the convergence criteria have been met. This supports the view of the CG method as an iterative method, while an iterative method often would not converge to the exact solution, especially in theory. Thus the CG method is sometimes treated as a semi-iterative method.

2.4 Preconditioning and convergence

2.4.1 Preconditioner for Linear Systems. As we discussed stationary iterative methods in subsection 2.3.1, we now review these methods from a preconditioning perspective.

The Jacobi and Gauss-Seidel iterations are of the form

$$\boldsymbol{u}^{k+1} = \boldsymbol{Q}\boldsymbol{u}^k + \boldsymbol{q} \tag{2.49}$$

in which

$$\boldsymbol{Q}_{JA} = \boldsymbol{I} - \boldsymbol{D}^{-1}\boldsymbol{A} \tag{2.50}$$

$$\boldsymbol{Q}_{GS} = \boldsymbol{I} - \boldsymbol{D} - \boldsymbol{L}^{-1}\boldsymbol{A} \tag{2.51}$$

for the Jacobi and Gauss-Seidel iterations, respectively. Given the matrix splitting

$$\boldsymbol{A} = \boldsymbol{S} - (\boldsymbol{S} - \boldsymbol{A}) \tag{2.52}$$

a *linear fixed-point iteration* can be defined by the recurrence

$$\boldsymbol{u}^{k+1} = \boldsymbol{S}^{-1}(\boldsymbol{S} - \boldsymbol{A})\boldsymbol{u}^k + \boldsymbol{S}^{-1}\boldsymbol{f} \tag{2.53}$$

which has the form Equation 2.49 with

$$\boldsymbol{Q} = \boldsymbol{S}^{-1}(\boldsymbol{S} - \boldsymbol{A}) = \boldsymbol{I} - \boldsymbol{S}^{-1}\boldsymbol{A}, \quad \boldsymbol{q} = \boldsymbol{S}^{-1}\boldsymbol{f} \tag{2.54}$$

For example, for the Jacobi iteration, $\boldsymbol{S} = \boldsymbol{D}, \boldsymbol{S} - \boldsymbol{A} = \boldsymbol{D} - \boldsymbol{A}$, while for the Gauss-Seidel iteration $\boldsymbol{S} = \boldsymbol{D} - \boldsymbol{L}, \boldsymbol{S} - \boldsymbol{A} = \boldsymbol{D} - \boldsymbol{L} - \boldsymbol{A} = \boldsymbol{U}$.

The iteration $\boldsymbol{u}^{k+1} = \boldsymbol{Q}\boldsymbol{u}^k + \boldsymbol{q}$ can also be viewed as a technique for solving the system

$$(\boldsymbol{I} - \boldsymbol{Q})\boldsymbol{u} = \boldsymbol{q} \tag{2.55}$$

Since $\boldsymbol{Q}$ has the form $\boldsymbol{Q} = \boldsymbol{I} - \boldsymbol{S}^{-1}\boldsymbol{A}$, this system can be rewritten as

$$\boldsymbol{S}^{-1}\boldsymbol{A}\boldsymbol{u} = \boldsymbol{S}^{-1}\boldsymbol{f} \tag{2.56}$$

We call this system that has the same solution as the original system $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{f}$ the *preconditioned system* and $\boldsymbol{S}$ the preconditioning matrix or preconditioner. It's often to use $\boldsymbol{M}$ to denote $\boldsymbol{S}$ when the splitting matrix $\boldsymbol{S}$ is used in the preconditioning. In other words, a *relaxation scheme* is equivalent to a *fixed-point iteration* on a *preconditioned system*. The preconditioning matrices can be easily derived for the Jacobi, Gauss-Seidel, SOR and SSOR iterations as follows

$$\boldsymbol{M}_{JA} = \boldsymbol{D} \tag{2.57}$$

$$\boldsymbol{M}_{GS} = \boldsymbol{D} - \boldsymbol{L} \tag{2.58}$$

$$\boldsymbol{M}_{SOR} = \frac{1}{\omega}(\boldsymbol{D} - \omega\boldsymbol{L}) \tag{2.59}$$

$$\boldsymbol{M}_{SSOR} = \frac{1}{\omega(2-\omega)}(\boldsymbol{D} - \omega\boldsymbol{L})\boldsymbol{D}^{-1}(\boldsymbol{D} - \omega\boldsymbol{U}) \tag{2.60}$$

The matrix $\boldsymbol{M}^{-1}$ should be symmetric and positive definite. Even though $\boldsymbol{M}$ is sparse most of the time, there is no guarantee that the $\boldsymbol{M}^{-1}$ is sparse. And this limits the number of techniques that can be applied to solve the preconditioned system. The computation of $\boldsymbol{M}^{-1}\boldsymbol{b}$ for any vector $\boldsymbol{b}$ should be small so the actual solution to the problem can be easily obtained from the solution to the preconditioned system.

2.4.2 General Convergence Results. In this section, we examine the convergence behaviors of the general preconditioners above. The detailed analysis can be found in Saad (2003). We choose the most important results to present here with notations adapted to match the other sections of the thesis. All methods seen in the previous section define a sequence of iterates of the form

$$\boldsymbol{u}^{k+1} = \boldsymbol{Q}\boldsymbol{u}^k + \boldsymbol{q} \tag{2.61}$$

where $\boldsymbol{Q}$ is the iteration matrix. We need to answer these two questions

– If the iteration converges, is the limit indeed a solution of the original system?

– Under which conditions does the iteration converge?

– How fast is the convergence?

If the above iteration converges to $\boldsymbol{u}$, and it satisfies

$$\boldsymbol{u} = \boldsymbol{Q}\boldsymbol{u} + \boldsymbol{q} \tag{2.62}$$

Recall the definition in Equation 2.54, it's easy to verify the $\boldsymbol{u}$ satisfies $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{f}$. This answers the first question. We now consider the next two questions.

If $\boldsymbol{I} - \boldsymbol{Q}$ is nonsingular, then there is a solution $\boldsymbol{u}^*$ to the equation Equation 2.62. Substracting Equation 2.62 from Equation 2.61 yields

$$\boldsymbol{u}^{k+1} - \boldsymbol{u}^* = \boldsymbol{Q}(\boldsymbol{u}^k - \boldsymbol{u}^*) = \cdots \boldsymbol{Q}^{k+1}(\boldsymbol{u}^0 - \boldsymbol{u}^*) \tag{2.63}$$

If the spectral radius of the iteration matrix $\boldsymbol{Q}$ is less than 1, then $\boldsymbol{u}^k - \boldsymbol{u}^0$ converges to zero. And the iteration Equation 2.61 converges toward the solution defined by Equation 2.62. Conversely, the relation

$$\boldsymbol{u}^{k+1} - \boldsymbol{u}^k = \boldsymbol{Q}(\boldsymbol{u}^k - \boldsymbol{u}^{k-1}) = \cdots \boldsymbol{Q}^k(\boldsymbol{q} - (\boldsymbol{I} - \boldsymbol{Q})\boldsymbol{u}^0)) \tag{2.64}$$

shows that if the iteration converges for any $\boldsymbol{u}^0$ and $\boldsymbol{q}$, then $\boldsymbol{Q}^k\boldsymbol{b}$ converges to zero for any vector $\boldsymbol{b}$. As a result, $\rho(\boldsymbol{Q})$ must be less than 1. The following theorem is proved:

Theorem 1. *Let $\boldsymbol{Q}$ be a square matrix such that $\rho(\boldsymbol{Q}) < 1$. Then $\boldsymbol{I} - \boldsymbol{Q}$ is non-singular and iteration Equation 2.61 converges for any $\boldsymbol{q}$ and $\boldsymbol{u}^0$. Conversely, if the iteration Equation 2.61 converges for any $\boldsymbol{q}$ and $\boldsymbol{u}^0$, then $\rho(\boldsymbol{Q}) < 1$*

The theorem and its proof can be found in Saad (2003) Since it is often expensive to compute the spectral radius of a matrix, sufficient conditions that guarantee convergence can be useful in practice. One such sufficient condition could be obtained by utilizing the inequality $\rho(\boldsymbol{Q} \leq ||\boldsymbol{Q}||)$ for any matrix norm Saad (2003).

Corollary 1.1. *Let $\boldsymbol{Q}$ be a square matrix such that $||\boldsymbol{Q}|| < 1$ for some matrix norm $||\cdot||$.. Then $\boldsymbol{I} - \boldsymbol{Q}$ is non-singular and the iteration Equation 2.61 converges for any initial vector $\boldsymbol{u}^0$*

Other than knowing that Equation 2.61 converges, we can also know how fast it converges. The error $\boldsymbol{e}^k = \boldsymbol{u}^k - \boldsymbol{u}^*$ at step k satisfies that

$$\boldsymbol{e}^k = \boldsymbol{Q}^k\boldsymbol{e}^0 \tag{2.65}$$

The proof can be found in Saad (2003)

The global asymptotic convergence factor is equal to the spectral radius of the iteration matrix. The general convergence rate differs from the specific rate only when the initial error does not have any components in the invariant subspace associated with the dominant eigenvalues. Since it is hard to know a priori, the general convergence factor is more useful in practice. The above analysis can be found in Saad (2003).

Using convergence analysis here, the convergence criteria for several iterative methods such as Richardson's Iteration, and regular splitting can be derived with simpler forms. One type of matrices that is worth notice is diagonally dominant matrices. We begin with a few standard definitions

Definition 3. *A matrix A is*

- *(weakly) diagonally dominant if*

$$|a_{j,j}| \geq \sum_{i=1, i\neq j}^{i=n} |a_{i,j}|, \quad j = 1, \ldots, n \tag{2.66}$$

- *strictly diagonally dominant if A is irreducible, and*

$$|a_{j,j}| > \sum_{i=1, i\neq j}^{i=n} |a_{i,j}|, \quad j = 1, \ldots, n \tag{2.67}$$

- *irreducibly diagonally dominant if*

$$|a_{j,j}| \geq \sum_{i=1, i\neq j}^{i=n} |a_{i,j}|, \quad j = 1, \ldots, n \tag{2.68}$$

  with strict inequality for at least one j

The diagonally dominant matrices are important as many matrices from the discretization of PDEs are diagonally dominant. When solving these linear systems with iterative methods, the spectral radius can be estimated using Gershgorin's theorem. Gershgorin's theorem allows determination of rough locations for all eigenvalues of $\boldsymbol{A}$. In situations where $\boldsymbol{A}$ is so large that the eigenvalues of $\boldsymbol{A}$ are unable to obtain, for example, a linear system from extremely fine discretization, the spectral radius can be directly obtained via the entries of the matrix $\boldsymbol{A}$. The simplest such result is the bound

$$|\lambda_i| \leq ||\boldsymbol{A}|| \tag{2.69}$$

for any matrix norm. Gershgorin's theorem provides a more precise localization result

Theorem 2 (Gershgorin)**.** *Any eigenvalue λ of a matrix $\boldsymbol{A}$ is located in one of the closed discs of the complex plane centered at $a_{i,i}$ and has the radius*

$$\rho_i = \sum_{j=1, j\neq i}^{j=n} |a_{i,j}| \tag{2.70}$$

In other words,

$$\forall \lambda \in \sigma(\boldsymbol{A}), \exists i \text{ such that} |\lambda - a_{i,i}| \leq \sum_{j=1, j\neq i}^{j=n} |a_{i,j}| \tag{2.71}$$

The theorem and its proof can be found in Saad (2003) This result also holds for the transpose of $\boldsymbol{A}$, so this theorem can also be formulated either based on column sums or row sums. The n discs defined in the theorem are called Gershgorin discs. The theorem states that the union of these n discs contains the spectrum of $\boldsymbol{A}$. It can also be shown that if there are m Gershgorin discs whose union S is disjoint from all other discs, then S contains exactly m eigenvalues (with multiplicities counted). Additional refinement which has important consequences concerns of a particular case when $\boldsymbol{A}$ is irreducible is given here.

Theorem 3. *Let $\boldsymbol{A}$ be an irreducible matrix and assume that an eigenvalue λ of $\boldsymbol{A}$ lies on the boundary of the union of the n Gershgorin discs. Then λ lies on the boundary of all Gershgorin discs*

This theorem and its proof can be found in Saad (2003) where an immediate corollary of the Gershgorin theorem and this theorem follows

Corollary 3.1. *If a matrix $\boldsymbol{A}$ is strictly diagonally dominant or irreducibly diagonally dominant, then it is nonsingular.*

This leads to the following theorem

Theorem 4. *If $\boldsymbol{A}$ is a strictly diagonally dominant or an irreducibly diagonally dominant matrix, then the associated Jacobi and Gauss-Seidel iterations converge for any $\boldsymbol{u}^0$.*

For SPD matrices, the convergence condition is given as follows

Theorem 5. *if $\boldsymbol{A}$ is symmetric with positive diagonal elements and for $0 < \omega < 2$, SOR converges for any $\boldsymbol{u}^0$ if and only if $\boldsymbol{A}$ is positive definite.*

These theorems for convergence play an important role in the study of various preconditioners and can be found in Saad (2003). More iterative methods and preconditioners as well as their convergence criteria can be found in the same book.

2.5 Multigrid Methods

The multigrid method is a scheme applied to solving a linear equation system with iterative solvers. It provides a convergence acceleration that improves the performance of these iterative solvers using grid coarsening Fedorenko (1973) Trottenberg, Oosterlee, and Schuller (2000). In practice, it can be implemented as a standalone method or as a preconditioner for other iterative methods such as the conjugate gradient method Tatebe (1993). Various multigrid methods are used in different branches of applied mathematics and engineering. such as electromagnetics Stolk, Ahmed, and Bhowmik (2014) and fluid dynamics Adler, Benson, Cyr, MacLachlan, and Tuminaro (2016).

The two important components of multigrid methods are the restriction and prolongation operators which transfer information between fine grids and coarse grids. These operators are typically based on linear interpolation procedures and are connected through variational properties Briggs, Henson, and McCormick (2000b) to ensure optimal coarse-grid correction in the A^h-norm with A^h being the left-hand side of the linear system defined on the fine grid. The multigrid method can be also applied to the SBP-SAT method with specific grid transfer operators. In this section, we will provide a brief review of the multigrid method and its implementation. We consider the following steady-state problem:

$$Lu = f, \text{ in } \Omega \tag{2.72}$$

$$Hu = g, \text{ on } \partial\Omega \tag{2.73}$$

where L is a differential operator on domain Ω, and H is a boundary operator on the boundary $\partial\Omega$. This is a generalization of many linear systems with various boundary conditions.

2.5.0.1 The multigrid algorithm. In general, the construction of a multigrid consists of the following four basic steps:

1. Fine-grid discretization
2. Error smoothing
3. Coarse-grid correction
4. Fine-grid update

Different combinations of these steps result in different multigrid schemes. The most simple scheme is a two-level multigrid V cycle. We will expand these four steps in the following sections.

2.5.0.2 Fine-grid discretization. Consider a *fine grid* meshing Ω_1 on Ω. A discrete linear system associated to Equation 2.72 on this fine grid Ω_1 has the general form

$$L_1\mathbf{u} = \mathbf{F} \tag{2.74}$$

where L_1 is the discrete version of the operator L in Equation 2.72 which also include boundary conditions in Equation 2.73. The vector $\mathbf{F}$ approximates f on the grid points of Ω_1 which already incorporates data for the boundary conditio**n** in Equation 2.73. $\mathbf{u}$ is the discretization of the solution u in the steady-state problem. We assume L_1 to be positive definite which also implies that L_1 is invertible. This is property is usually satisfied from discretization methods.

2.5.0.3 Error smoothing. Error smoothing is required prior to grid coarsening. Suppose we have an initial guess $\mathbf{u}^0$, the iterative approach towards the solution to Equation 2.74 is through solving

$$\mathbf{w}_\tau + L_1\mathbf{w}(\tau) = \mathbf{F},\ 0 < \tau < \Delta\tau \tag{2.75}$$

$$\mathbf{w}(0) = \mathbf{u}^{(0)} \tag{2.76}$$

where $\Delta_t > 0$ is the *smoothing step*. The solution to this equation is

$$\mathbf{w}(\Delta\tau) = e^{-L_1\Delta\tau}\mathbf{u}^0 + (I_1 - e^{-L_1\Delta\tau})L_1^{-1}\mathbf{F} \tag{2.77}$$

where I_1 is the identity matrix on Ω_1, and the following condition holds for any norm if L_1 is positive definite

$$||\mathbf{w}(\Delta\tau) - \mathbf{u}|| < ||\mathbf{u}^{(0)} - \mathbf{u}|| \tag{2.78}$$

Smoothing technique for the solution can be defined as follows

$$\begin{aligned} \mathbf{w}^k &= S\mathbf{w}^{k-1} + (I_1 - S)L_1^{-1}\mathbf{F}, k = 1, \dots \nu \\ \mathbf{w}^0 &= \mathbf{u}^0 \end{aligned} \tag{2.79}$$

where S is the smoother. If S is an exponential smoother $S_{\exp} = e^{-L_1 \Delta\tau}$, this will yield the pseudo time-marching procedure in Equation 2.75. This iterative method would converge after ν steps to

$$\mathbf{w} = S^{\nu}\mathbf{u}^0 + (I_1 - S^{\nu})L_1^{-1}\mathbf{F} \tag{2.80}$$

The convergence criteria for this procedure is mentioned in the overview of iterative methods.

2.5.0.4 Coarse-grid correction. Next, consider the error $\mathbf{e} = L_1^{-1}\mathbf{F} - \mathbf{w}$ and the residual problem

$$L_1\mathbf{e} = \mathbf{F} - L_1\mathbf{w} \tag{2.81}$$

Instead of solving this system directly, we introduce a subset of Ω_1 called the *coarse grid* Ω_2, and solve the associated coarse grid problem on Ω_2

$$L_2\mathbf{d} = I_r(\mathbf{F} - L_1\mathbf{w}) \tag{2.82}$$

This problem is obtained from the finer grid problem Equation 2.81 by using the following operators

1. a restriction operator $I_r : \Omega_1 \to \Omega_2$
2. a coarse-grid operator L_2: $\Omega_2 \to \Omega_2$

The coarse-grid operator can be built by using the Galerkin condition

$$L_2 = I_r L_1 I_p \tag{2.83}$$

where $I_p \ : \Omega_2 \to \Omega_1$ is a the prolongation operator. In some situations, L_2 can be built independently through the direct use of discretization methods, but I_r and I_p needs to be carefully defined so the Galerkin condition Equation 2.83 still holds.

The prolongation operator I_p is commonly chosen through linear interpolation. Assume Ω_1 had a grid spacing of $\Delta x = 1/N$, and Ω_2 consists of the even grid points of Ω_1. This leads to

$$(I_p\mathbf{v})_m = \begin{cases} v_j, & m = 2j, j = 0, \ldots, N/2 \\ \frac{1}{2}(v_j + v_{j+1}), & m = 2j+1, j = 0, \ldots, N/2 \end{cases} \tag{2.84}$$

As we already define the prolongation operator, the restriction operator is given as

$$I_r = I_p^T/C \tag{2.85}$$

which is called the variational property. The C is a constant determined by the discretization method. In this problem, the value for C is 2.

2.5.0.5 Fine-grid update. Finally, we update the fine grid solution with correction **d** as

$$\mathbf{u}^{(1)} = \mathbf{w} + I_p \mathbf{d} \tag{2.86}$$

The relation Equation 2.86, together with Equation 2.80 and Equation 2.82 provides an iterative method for solving the steady-state problem

$$\mathbf{u}^{n+1} = M\mathbf{u}^n + N\mathbf{F} \tag{2.87}$$

where

$$M = CS^\nu \tag{2.88}$$

$$C = I_1 - I_p L_2^{-1} I_r L_1 \tag{2.89}$$

$$N = (I_1 - M)L_1^{-1} \tag{2.90}$$

M is called the multigrid iteration matrix here and C is referred to as the coarse grid correction operator. Here, M plays a central role in the convergence of the iterative method. We can see this by the definition of the error at step n $\mathbf{e}^{(n)} = \mathbf{u}^{(n)} - L_1^{-1}\mathbf{F}$ as we get

$$\mathbf{e}^{(n+1)} = M\mathbf{e}^{(n)} \tag{2.91}$$

which again leads to the same convergence criteria for the iterative method depending on the spectral radius of M.

To demonstrate the actual process of a multigrid scheme, we use the following two-grid correction scheme as an example

1. Relax ν_1 times on $L_1^h \mathbf{u}^{(1)} = \mathbf{F}^{(1)}$ on Ω_1 with the initial guess $\mathbf{v}^1$
2. Compute the fine-grid residual $\mathbf{r}^{(1)} = \mathbf{F}(1) - L_1\mathbf{v}^{(1)}$ and restrict it to the coarse grid by $\mathbf{r}^{(2)} = I_r\mathbf{r}^{(1)}$
3. Solve $L_2\mathbf{e}^{(2)} = \mathbf{r}^{(2)}$ (or relax ν_1 times) on Ω_2
4. Interpolate the coarse-grid error to the fine grid by $\mathbf{e}^{(1)} = I_p\mathbf{e}^{(2)}$ and correct the fine-grid approximation by $\mathbf{v}^1 \leftarrow \mathbf{v}^{(1)} + \mathbf{e}^{(1)}$
5. Relax ν_2 times on $L_1\mathbf{u}^{(1)} = \mathbf{F}^{(1)}$ on Ω_1 with the initial guess $\mathbf{v}^{(1)}$

There are more schemes for multigrid, and the main schemes are summarized inFigure 1.

Earlier work in multigrid relies on the geometric structure to construct coarse problems, thus this approach is called geometric multigrid. In problems where the computational domain is not composed of well-structured meshes, the multigrid method can be also applied via algebraic operators rather than a geometric grid. This approach is called the algebraic multigrid. We will cover this approach in the next subsection.

2.5.1 Algebraic Multigrid. The classical multigrid formed around the geometric structure has been generalized that the multigrid is analyzed in terms of the matrix properties McCormick and Ruge (1982). This algebraic approach to theory was further extended to form the basis for much of the early development that led to the so-called Ruge-Stüben or classical algebraic multigrid (CAMG) method Brandt (1986); Mandel (1988); Ruge and Stüben (1987). A detailed overview of the algebraic multigrid can be found in this recent paper Xu and Zikatanov (2017). Here, we want to present it more concisely. We begin this subsection with the following theorem in linear algebra taken from Briggs et al. (2000a).

Theorem 6 (Solvability and the Fundamental Theorem of Linear Algebra)**.** *Suppose we have a matrix $A \in \mathbf{R}^{m\times n}$. The fundamental theorem of linear algebra states that the range (column space) of the matrix, $\mathcal{R}(A)$, is equal to the orthogonal complement of $\mathcal{N}(A^T)$, the null space of A^T. Thus, spaces $\mathbf{R}^m$ and $\mathbf{R}^n$ can be orthogonally decomposed as follows:*

$$\mathbf{R}^m = \mathcal{R}(A) \oplus \mathcal{N}(A^T) \tag{2.92}$$

$$\mathbf{R}^n = \mathcal{R}(A) \oplus \mathcal{N}(A) \tag{2.93}$$

For the equation $A\mathbf{u} = \mathbf{f}$ to have a solution, it is necessary that the vector $\mathbf{f}$ lie in $\mathcal{R}(A)$. Thus, an equivalent condition is that $\mathbf{f}$ be orthogonal to every vector in $\mathcal{N}(A^T)$. For the equation $A\mathbf{u} = \mathbf{f}$ to have a unique solution, it is necessary that $\mathcal{N}(A) = \{\mathbf{0}\}$. Otherwise, if $\mathbf{u}$ is a solution and $\mathbf{v} \in (A)$, then $A(\mathbf{u} + \mathbf{v}) = A\mathbf{u} + A\mathbf{v} = \mathbf{f} + \mathbf{0} = \mathbf{f}$, so the solution is not unique.

This is another point to view the coarse-grid correction scheme, and this leads to the algebraic multigrid. More theories related to this topic and the spectral picture of multigrid can be found in Briggs et al. (2000a).

The unique aspect of the CAMG is that the coarse problem is defined on a subset of the degrees of freedom of the initial problem, thus resulting in both coarse and fine points, which

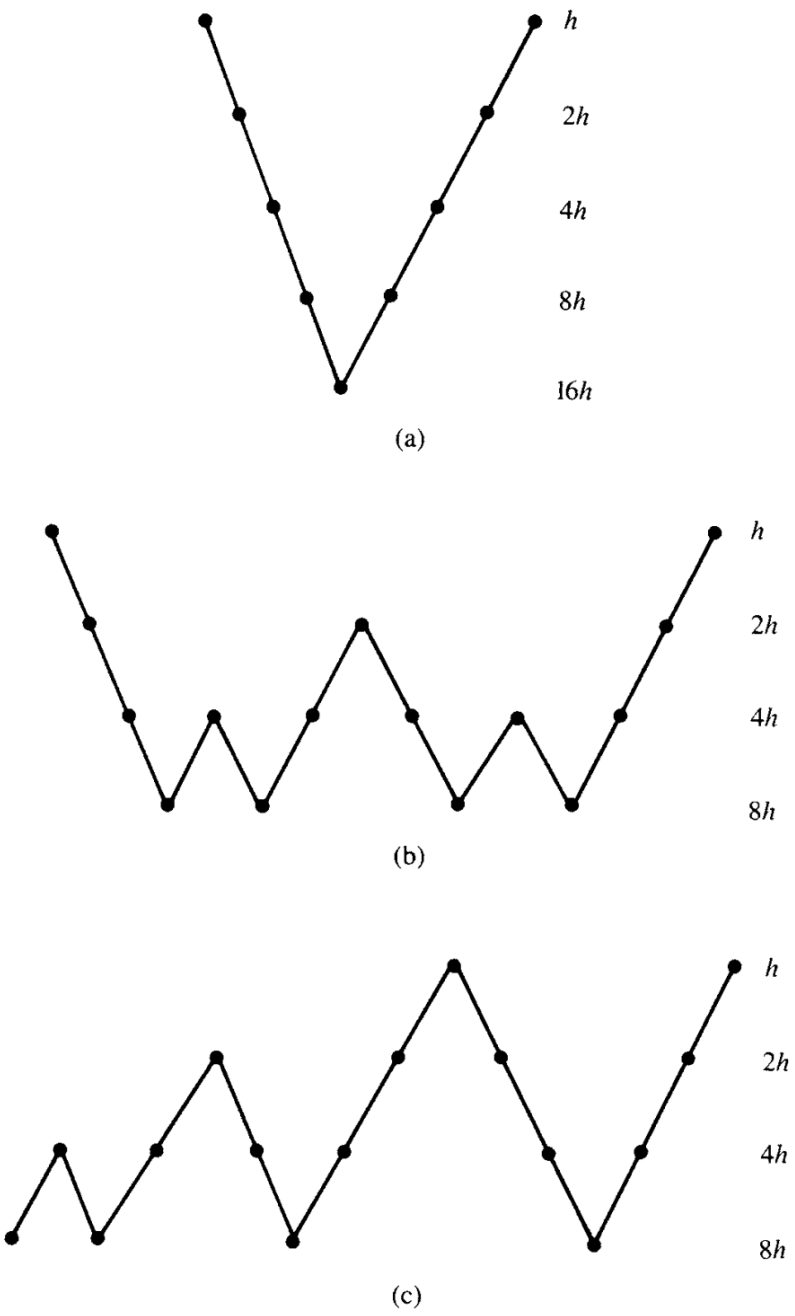

Figure 1. Schedule of grids for (a) V-cycle, (b) W-cycle, and (c) FMG scheme, all on four levels. Briggs et al. (2000a)

leads to the term CF-based AMG. A different approach to constructing algebraic multigrid is called *smoothed aggregation* AMG (SA), where collections of degrees-of-freedom define a coarse degree-of-freedom Vaněk, Mandel, and Brezina (1996). Together, CF and SA form the basis of AMG and led to several developments that extend AMG to a wider class of problems and architectures.

AMG does not depend on the geometry of the problem and discretization schemes, and due to this generalizability, it has been implemented in different forms in many software libraries. The original CAMG algorithm and its variants are available as `amg1r5` and `amg1r6` Ruge and Stüben (1987). A parallel implementation of the CF-based AMG can be found in the BoomerAMG package in the Hypre library Yang et al. (2002). The Trilinos package includes `ML` as a parallel SA-based AMG solver Gee, Siefert, Hu, Tuminaro, and Sala (2006). Finally, PyAMG includes a number of AMG variants for testings, and Cusp distributes with a standard SA implementation for use on a GPU Bell, Olson, and Schroder (2022); Dalton, Bell, Olson, and Garland (2014).

2.5.2 The Multigrid Method Within the SBP-SAT Scheme. Since the SBP-SAT scheme is a framework for discretization to form a linear system, it is compatible with the multigrid method and can be accelerated using this technique. The key challenge from simply applying the common prolongation and restriction operators with the Galerkin condition Equation 2.83 is that the summation-by-parts property would not be preserved for the coarse grid operators. In order to accurately represent the coarse-grid correction problem for the SBP-SAT scheme, a more suitable class of interpolation operators needs to be proposed. Many works have been done to address this issue Ruggiu, Weinerfelt, and Nordström (2018); Ruggiu, Weinerfelt, and Nordström (2018).

To overcome this issue, consider defining the restriction operator as

$$I_r = H_2^{-1} I_p^T H_1 \tag{2.94}$$

which was first introduced in Ruggiu, Weinerfelt, and Nordström (2018). This involves the coarse grid SBP norm H_2 and is obtained by enforcing that two scalar products

$$(\phi_1, \psi_1)_{H_1} = (\phi_1 H_1 \psi_1) \tag{2.95}$$

$$(\phi_2, \psi_2)_{H_2} = (\phi_2 H_2 \psi_2) \tag{2.96}$$

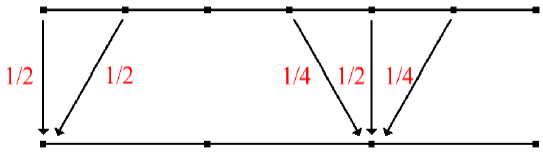

Figure 2. The 2nd-order SBP-preserving restriction operator I_r

are equal for $\phi_1 = I_p\phi_2$ and $\psi_2 = I_r\psi_1$. ϕ and ψ correspond to the $\boldsymbol{u}$ and $\boldsymbol{v}$ in the previous section that describes the SBP-SAT methods in detail. We use these new notations to avoid confusion with the $\mathbf{u}$ used in the previous subsection on the multigrid method. As a result, the interpolation operators I_r and I_p are adjoints to each other with respect to the SBP-based scalar products defined in Hackbusch (2013b).

$$(I_p, \xi_2, \xi_1)_{H_1} = (\xi_2, I_r\xi_1)_{H_2} \tag{2.97}$$

by using Equation 2.94, it is possible to build pairs of consistent and accurate prolongation and restriction operators. The following definition of the SBP-preserving interpolation operators was given in Ruggiu, Weinerfelt, and Nordström (2018), where the operators were used to couple SBP-SAT formulations on grids with different mesh sizes with numerical stability.

Definition 4. *Let the row-vectors* $\mathbf{x}_1^k$ *and* $\mathbf{x}_2^k$ *be the projections of the monomial* x^k *onto equidistant 1-D grids corresponding to a fine and coarse grid, respectively.* I_r *and* I_p *are then called 2q-th order accurate SBP-preserving interpolation operators if* $I_r\mathbf{x}_1^k - \mathbf{x}_2^k$ *and* $I_p\mathbf{x}_2^k - \mathbf{x}_1^k$ *vanish for* $k = 0, ..., 2q-1$ *in the interior and for* $k = 0, ..., q-1$ *at the boundaries.*

The sum of the orders of the prolongation and restriction operators should be at least equal to the order of the differential equation. As a consequence, the use of high-order interpolation is not required here to solve the linear system with the multigrid method. However, high-order grid transfer operators can be used in combination with high-order discretization Sundar, Stadler, and Biros (2015).

SBP-preserving interpolation operators with minimal bandwidth are given in Appendix A. The restriction operator I_r, which differs from the conventional one at boundary nodes, is shown in Figure 2.

2.5.2.1 *SBP-preserving interpolation applied to the first derivative.*

Using Galerkin condition Equation 2.83 and SBP-preserving operators, we can construct the linear system with the multigrid method. We first consider the first derivative fine-grid SBP operator $D_{1,1}$ and its coarse-grid counterpart $D_{1,2}$ constructed as follow

$$D_{1,2} = I_r D_{1,1} I_p \tag{2.98}$$

We now show that $D_{1,2}$ preserves SBP property in such ways. To start with, we rewrite the left-hand side of the following SBP property

$$(\phi, D_1\psi)_H = \phi_N\psi_N - \phi_0\psi_0 - (D_1\phi, \psi)_H \tag{2.99}$$

with the adjoint relation Equation 2.97 as follows

$$\begin{aligned}(\phi_2, D_{1,2}\psi_2)_{H_2} &= (\phi_2, I_r(D_{1,1}I_p\phi_2))_{H_2} \\ &= (I_p\phi_2, D_{1,1}(I_p\psi_2))_{H_1}\end{aligned} \tag{2.100}$$

Next, the SBP property for the finite-grid operator D_1 leads to

$$\begin{aligned}(\phi_2, D_{1,2}\psi_2)_{H_2} &= (I_p, \phi_2)_N(I_p, \psi_2)_N \\ &\quad - (I_p, \phi_2)_0(I_p, \psi_2)_0 - (D_{1,1}(I_p\phi_2), I_p\psi_2)_{H_1}\end{aligned} \tag{2.101}$$

Both grids are conforming to the domain boundaries, and the prolongation onto the boundary nodes of the fine grid is exact. Furthermore, by applying Equation 2.97 to the right-hand side of Equation 2.101, we obtain

$$(\phi_2, D_{1,2}\psi_2)_{H_2} = \phi_{2,N/2}\psi_{2,N/2} - \phi_{2,0}\psi_{2,0} - (D_{1,2}\phi_2, \psi_2)_{H_2} \tag{2.102}$$

And we've shown that the coarse grid operator $D_{1,2}$ constructed in a such way preserves the SBP property. Also, the coarse grid first derivative SBP operator $D_{1,2}$ retains the order of accuracy of the original scheme at the interior nodes if $2q$th order SBP-preserving interpolations are used. The proof can be found in Ruggiu, Weinerfelt, and Nordström (2018).

2.5.2.2 *SBP-preserving interpolation applied to the second derivative.*

The SBP-preserving interpolation can also be applied to the second derivative operator. Similar to the proof for the first derivative operator, we can prove that the coarse grid operator constructed in such ways preserves the SBP property.

The interpolation operators in Equation 2.94 lead to a coarse-grid second derivative operator $D_{2,2}$ which preserves the summation-by-parts property in Equation 2.4. We can show that by rewriting the left hand side of the Equation 2.4 for $D_{2,2}$ and the two coarse-grid functions ϕ_2 and ψ_2 by using Equation 2.97.

$$(\phi_2, D_{2,2}\psi_2)_{H_2} = (\phi_2, I_r, D_{2,1}I_p\psi_2)_{H_2} = (I_p\phi_2, D_{2,1}I_p\psi_2)_{H_1} \tag{2.103}$$

By applying the SBP property Equation 2.4 for the fine-grid second derivative $D_{2,1}$, we have

$$\begin{aligned}(\phi_2, D_{2,2}\psi_2)_{H_2} &= (I_p\phi_2)_N(SI_p\psi_2)_N \\ &\quad - (I_p\phi_2)_0(SI_p\psi_2)_0 - (SI_p\phi_2)^T A(SI_p\psi_2)_{H_2}\end{aligned} \tag{2.104}$$

Both grids are conforming to domain boundaries, implying that $(I_p\phi_2)_i = \phi_{2,i/2}$ and $(SI_p\psi_2) = (S\phi_2)_{i/2}$ for $i \in \{0, N\}$. Thus

$$\begin{aligned}(\phi_2, D_{2,2}\psi_2)_{H_2} &= \phi_{2,N/2}(S\phi_2)_{N/2} \\ &\quad - \phi_{2,0}(S\phi_2)_0 - (SI_p\phi_2)^T A(SI_p\psi_2)_{H_2}\end{aligned} \tag{2.105}$$

where S is equivalent to $\boldsymbol{d}_0^T$ in **??** which approximates the first derivative at the boundaries.

Additional proofs or propositions to SBP-preserving interpolations can also be found in Ruggiu, Weinerfelt, and Nordström (2018). Furthermore, several model problems have been tested with multigrid iteration schemes using these SBP-preserving interpolations. These problems include a Poisson equation, the anisotropic elliptic problem, and the advection-diffusion problem. Numerical experiments show that the SBP-preserving interpolation improves convergence properties of the multigrid scheme for SBP-SAT discretizations regardless of the order of the discretization and smoother chosen. Moreover, the excellent performance in combination with the smoother SOR, clearly indicates that multigrid algorithms with SBP-preserving interpolation can be designed to get fast convergence. The paper mainly covers the steady model problem to compare the effect of different grid transfer operators. For time-dependent problems, the effectiveness of multigrid algorithms with these SBP-preserving interpolations needs to be tested Ruggiu, Weinerfelt, and Nordström (2018).

2.5.3 Multigrid Preconditioned Conjugate Gradient. In the previous sections, we introduce the classical iterative solvers and Krylov subspace methods as a solver. Moreover, we show that the classical iterative solvers can be used in the multigrid method as smoothers. And we provide the basic knowledge on preconditioners for iterative methods. However, using multigrid

as a preconditioner for the conjugate gradient is an alternative approach motivated by engineering problems.

The multigrid method is a very effective iterative method for the mechanical analysis of heterogeneous material samples in Häfner, Eckardt, Luther, and Könke (2006). However, the increase in the ratio of Young's moduli between matrix material and inclusion leads to a significantly worse condition number of the system, which slows the solution process. This could be also the result of the worse material representation on coarse grids. For a similar problem, Poisson's equation with large coefficient jumps or differences of grid spacing in coordinate transformation, the worse condition number will also lead to the slow solving process with the iterative methods mentioned above. As Poisson's equation is the key challenge in earthquake cycle simulation, an effective approach to solving linear systems with worse condition numbers is worth exploring. It has been shown that the multigrid preconditioned conjugate gradient method has a superior convergence rate over the multigrid method as a solver Tatebe (1993). This approach is less dependent on the considered problem.

The conditions of the multigrid preconditioners are examined in Tatebe (1993). According to Wesseling (2004), the multigrid method will potentially provide a valid preconditioner if the smoother is symmetric. For a derivation of the preconditioned conjugate gradient method, we would introduce a matrix $\boldsymbol{L}$ which satisfies $\boldsymbol{M}^{-1} = \boldsymbol{L}^T\boldsymbol{L}$ as shown in Wesseling (2004) (Our notation $\boldsymbol{M}$ is equivalent to $\boldsymbol{H}$ in the paper). The Equation 2.56 improves the convergence if the condition number of the preconditioned matrix $\boldsymbol{M}^{-1}\boldsymbol{A}$ is lower than that of the original matrix $\boldsymbol{A}$, which can be determined from the analysis of eigenvalues as presented in Hackbusch (2013a). If the preconditioning matrix is exactly $\boldsymbol{M}^{-1} = \boldsymbol{A}^{-1}$, the after one iteration step, the exact solution $\boldsymbol{u}$ is found. An ideal preconditioning matrix $\boldsymbol{M}^{-1}$ should be a reasonably close approximation of $\boldsymbol{A}^{-1}$. With respect to the initial search direction, the vector $\boldsymbol{p}^0 = \boldsymbol{M}^{-1}\boldsymbol{r}^0$ would correspond to the error $-\boldsymbol{e}^0$, if $\boldsymbol{M}^{-1} = \boldsymbol{A}^{-1}$. An adequate matrix $\boldsymbol{M}^{-1}$ leads to an improved initial search direction $\boldsymbol{p}^0$. Therefore, the preconditioned conjugate gradient method applies the following start conditions

$$\boldsymbol{r}^0 = \boldsymbol{f} - \boldsymbol{A}\boldsymbol{u}^0; \quad \tilde{\boldsymbol{r}}^0 = \boldsymbol{p}^0 = \boldsymbol{A}^{-1}\boldsymbol{r}^0 \tag{2.106}$$

The following equations give a preconditioned conjugate gradient method adapted from Tatebe (1993) in the notation of the conjugate gradient method in subsection 2.3.2.

$$\lambda_k = \frac{\tilde{\boldsymbol{r}}^{k^T}\boldsymbol{r}^k}{\boldsymbol{p}^{k^T}\boldsymbol{A}\boldsymbol{p}^k} \tag{2.107}$$

$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \lambda_k \boldsymbol{p}^k \tag{2.108}$$

$$\boldsymbol{r}^{k+1} = \boldsymbol{r}^k - \lambda_k \boldsymbol{A}\boldsymbol{p}^k \tag{2.109}$$

$$\tilde{\boldsymbol{r}}^{k+1} = \boldsymbol{M}^{-1}\boldsymbol{r}^{k+1} \tag{2.110}$$

$$\boldsymbol{p}^{k+1} = \tilde{\boldsymbol{r}}^{k+1} + \frac{\tilde{\boldsymbol{r}}^{k+1^T}\boldsymbol{r}^{k+1}}{\tilde{\boldsymbol{r}}^{k^T}\boldsymbol{r}^k}\boldsymbol{p}^k \tag{2.111}$$

In each iteration step, preconditioning only takes place in Equation 2.110 and generates a new vector $\tilde{\boldsymbol{r}}^{k+1}$. The preconditioning matrix $\boldsymbol{M}^{-1}$ does not need to be explicitly built. The operation defined in Equation 2.110 can be replaced by a multigrid cycle that solves a linear system with $\boldsymbol{r}^{k+1}$ being the right hand side, and the solution is then assigned to $\tilde{\boldsymbol{r}}^{k+1}$. The preconditioned conjugate gradient method preserves a system of conjugate directions, while each increment is optimized for each improved search direction based on the multigrid method. Therefore, this optimization leads to considerably improved increments, if the stiffness of the coarse meshes is generally overestimated.

2.6 Geometric multigrid for SBP-SAT method

To apply multigrid efficiently for the SBP-SAT method on GPUs, we need to develop a new geometric multigrid formulation that does not require using algebraic coarsening or Galerkin's condition.

Matrix-free iterative methods enable the solution to larger problems compared to a direct solve that requires storing a matrix factorization. However, the convergence of CG depends predominantly on the condition number and quality of the initial guess. The condition number can be reduced through preconditioning techniques, but the preconditioning matrix $\mathbf{M}$ has to be SPD and fixed, and although it need not be explicitly assembled nor inverted, good preconditioners should satisfy $\mathbf{M} \approx \mathbf{A}^{-1}$.

To our knowledge, preconditioning has not been explored for CG methods applied to SBP-SAT discretizations. Existing work using multigrid as a solver for problems with SBP-SAT methods focused on using SBP-preserving interpolation operators with the Galerkin coarsening to build the coarse grid operators Ruggiu, Weinerfelt, and Norström (2018). Here the standard

interpolation operators were modified for boundary points to preserve the SBP property Ruggiu, Weinerfelt, and Norström (2018). However, although Galerkin coarsening or other algebraic multigrid methods produce coarse grid operators automatically (and therefore can be seen as a "plug-in" solver for any linear system Stüben (2001)), defining these matrix-free coarse grid operators in this fashion requires writing a different kernel for every grid level, as well as more memory for data storage Brandt (2006). Moreover, it also increases overhead in pre-compiling matrix-free kernels for different Ns due to the just-in-time (JIT) compiling mechanism in Julia. Therefore, to fully utilize the efficiency of our matrix-free methods, as well as to reduce complexity in and number of kernels needed, developing geometric multigrid preconditioned CG (denoted MGCG) for the SBP-SAT method becomes the key focus of our work.

Three key ingredients define multigrid methods, namely, interpolation operators (prolongation and restriction), smoothers, and (if used) a direct solve on a coarse grid. In this work we adopt the second-order SBP-preserving prolongation/restriction operators from Ruggiu, Weinerfelt, and Norström (2018), which maintain accuracy at domain boundaries and correctly transfer residual vectors to the coarser grids. The 2D restriction operator is given by

$$\boldsymbol{I}_h^{2h} = \boldsymbol{H}_{2h}^{-1} \left(\boldsymbol{I}_{2h}^h\right)^T \boldsymbol{H}_h \tag{2.112}$$

where $\boldsymbol{H}_h$ and $\boldsymbol{H}_{2h}$ denote $\boldsymbol{H} \otimes \boldsymbol{H}$ with grid spacing h and $2h$, respectively. The 2D prolongation operator $\boldsymbol{I}_{2h}^h$ is defined by $\boldsymbol{I}_{2h}^h = I_{2h}^h \otimes I_{2h}^h$, where I_{2h}^h is the standard 1D prolongation operator Briggs, Henson, and McCormick (2000c), see Appendix A in Ruggiu, Weinerfelt, and Norström (2018).

One feature that differentiates our problem formulation from those in Ruggiu, Weinerfelt, and Norström (2018) is that our matrix in (4.9) is rendered SPD only after the multiplication of (2.5) on the left by $(\boldsymbol{H} \otimes \boldsymbol{H})$, which introduces additional grid information when calculating the associated residual vector. To properly transfer this grid information we found improved performance when further modifying the restriction operator to account for grid spacing. This is achieved by excluding the $(\boldsymbol{H} \otimes \boldsymbol{H})$ term when computing the residual on the fine grid, then restricting using $\boldsymbol{I}_r$, and then re-introducing the grid spacing on the coarse grid. This process requires "applying" the inverse: Given that $(\boldsymbol{H} \otimes \boldsymbol{H})$ is a sparse diagonal matrix (thus its inverse is the diagonal matrix of reciprocal values), GPU kernels for the multiplication of this matrix and

its inverse can be easily implemented in a matrix-free manner. The pseudo-code for the geometric multigrid method is given in Algorithm 1.

Algorithm 1 $(k+1)$-level MG for $\boldsymbol{A}_h \boldsymbol{u}_h = \boldsymbol{f}_h$, with smoothing $S_{h_k}^{\nu}$ applied ν times. SBP-preserving restriction and interpolation operators are applied. Grid coarsening ($k \rightarrow k+1$) is done through successive doubling of grid spacing until reaching the coarsest grid. The multigrid cycle can be performed $N_{maxiter}$ times. $\boldsymbol{r}$ represents the residual, and $\boldsymbol{v}$ represents the solution to the residual equation used during the correction step. This algorithm is adapted from Liu and Henshaw (2023).

function MG($\boldsymbol{f}_{h_k}$, $\boldsymbol{A}_{h_k}$, $\boldsymbol{u}_{h_k}^{(0)}$, k, $N_{maxiter}$)
  for $n = 0, 1, 2, \ldots, N_{maxiter}$ **do**
    $\boldsymbol{u}_{h_k}^{(n)} \xleftarrow{S_{h_k}^{\nu_1}} \boldsymbol{u}_{h_k}^{(n)}$ ▷ Pre-smoothing ν_1 times
    $\boldsymbol{r}_{h_k}^{(n)} = \boldsymbol{f}_{h_k}^{(n)} - \boldsymbol{A}_{h_k}^{(n)} \boldsymbol{u}_{h_k}^{(n)}$ ▷ Calculating residual
    $\tilde{\boldsymbol{r}}_{h_k} = (\boldsymbol{H}_k \otimes \boldsymbol{H}_k)^{-1} \boldsymbol{r}_{h_k}^{(n)}$ ▷ Removing grid info
    $\boldsymbol{r}_{h_{k+1}} = (\boldsymbol{H}_{k+1} \otimes \boldsymbol{H}_{k+1}) \boldsymbol{I}_{h_k}^{h_{k+1}} \tilde{\boldsymbol{r}}_{h_k}$ ▷ Restriction
    if $k+1 = k_{\max}$ **then**
      $\boldsymbol{v}_{h_{k+1}}^{(n)} \xleftarrow{S_{h_{k+1}}^{\nu_2}} \boldsymbol{0}_{h_{k+1}}$ ▷ Smoothing on coarsest grid
    else
      $\boldsymbol{v}_{h_{k+1}}^{(n)} = \mathrm{MG}(\boldsymbol{r}_{h_{k+1}}, \boldsymbol{A}_{h_{k+1}}, \boldsymbol{0}_{h_{k+1}}, k+1, 1)$
      ▷ Recursive definition of MG
    end if
    $\boldsymbol{v}_k^n = \boldsymbol{I}_{h_{k+1}}^{h_k} \boldsymbol{v}_{k+1}^{(n)}$ ▷ Interpolation
    $\boldsymbol{u}_k^{(n+1)} = \boldsymbol{u}_k^{(n)} + \boldsymbol{v}_k^n$ ▷ Correction
    $\boldsymbol{u}_k^{(n+1)} \xleftarrow{S_{h_k}^{\nu_3}} \boldsymbol{u}_{h_k}^{(n+1)}$ ▷ Post-smoothing ν_3 times
  end for
end function

Many types of smoothers for multigrid methods can be explored for best performance. In this work we choose the Richardson iteration given by $\mathbf{x}_{k+1} = \mathbf{x}_k + \omega(\mathbf{b} - \mathbf{A}\mathbf{x}_k)$ because it can be easily implemented with our existing matrix-free kernel. Here ω is chosen to satisfy the convergence criteria and its optimal value depends on the eigenvalues of $\boldsymbol{A}$ as $\omega_{opt} = \frac{2}{\lambda_{max} + \lambda_{min}}$, where λ_{max} and λ_{min} are the largest and smallest eigenvalues of $\boldsymbol{A}$ respectively. We use Arpack.jl, which is a Julia wrapper of ARPACK that uses the Implicitly Restarted Arnoldi Method to calculate eigenvalues for sparse matrices. For small N, we can compute λ_{max} and λ_{min} directly, but for large N values, these become computationally intractable. We use interpolation to approximate values for λ_{max} and λ_{min} for $N \geq 32$ based on observation of eigenvalues for $N \leq 32$,

namely,

$$\lambda_{min,2N} = \lambda_{min,N}/4,$$

$$\lambda_{max,2N} = \lambda_{max,N} + 0.6 * (\lambda_{max,N} - \lambda_{max,N/2}),$$

where $\lambda_{min,N}$ represents the minimal eigenvalue of a linear system formed for our 2D problem with $N + 1$ grid points in each direction and $\lambda_{max,N}$ is the corresponding maximum value. In practice, these interpolated eigenvalues provide a relatively tight lower and upper bound for the real eigenvalues and appear to be sufficient according to our performance results. Alternative smoothers could be considered, such as Jacobi iteration or SSOR, but these require the decomposition of the linear system and the development of additional GPU kernels. We did test these smoothers in experiments using the matrix-explicit formulation and found that they perform at similar levels to the Richardson iteration when multigrid is used as a preconditioner. We found that the total number of iterations required by MGCG is largely determined by the number of grid levels and smoothing steps and is less impacted by the choice of smoother itself.

Multigrid methods have many tunable parameters. For this initial study, we implemented MGCG with 5 Richardson pre- and post-smoothing steps on every level with a single V-cycle (i.e. taking $\nu_1 = \nu_2 = \nu_3 = 5$ and $N_{maxiter} = 1$ in Algorithm 1), including on the coarsest grid (5 grid points in each direction). This avoids using a direct solve on the coarse grid which would require conversion between CPU arrays and GPU arrays. All operations in this MGCG algorithm can be implemented in a matrix-free manner in a way that does not require storing matrix $\boldsymbol{A}$ on any grid level.

To show the drastically different behaviors, we plot the discrete L^2-error against iteration counts for $N = 32$ for CG and MGCG in Figure 3. MGCG converges after only $\sim$5 iterations. Additional iterations are coming from the additional discrete L^2-error requirement in the stopping criteria. Since the complexity of each CG iteration step is $\mathcal{O}(N^2)$, as N doubles the total time increases by a factor of 4 for MGCG versus 8 for CG. In this section we present a new formulation multigrid preconditioned conjugate gradient that can be implemented matrix-free with a Richardson smoother in order to solve 2D, variable coefficient elliptic problems discretized with an SBP-SAT method.

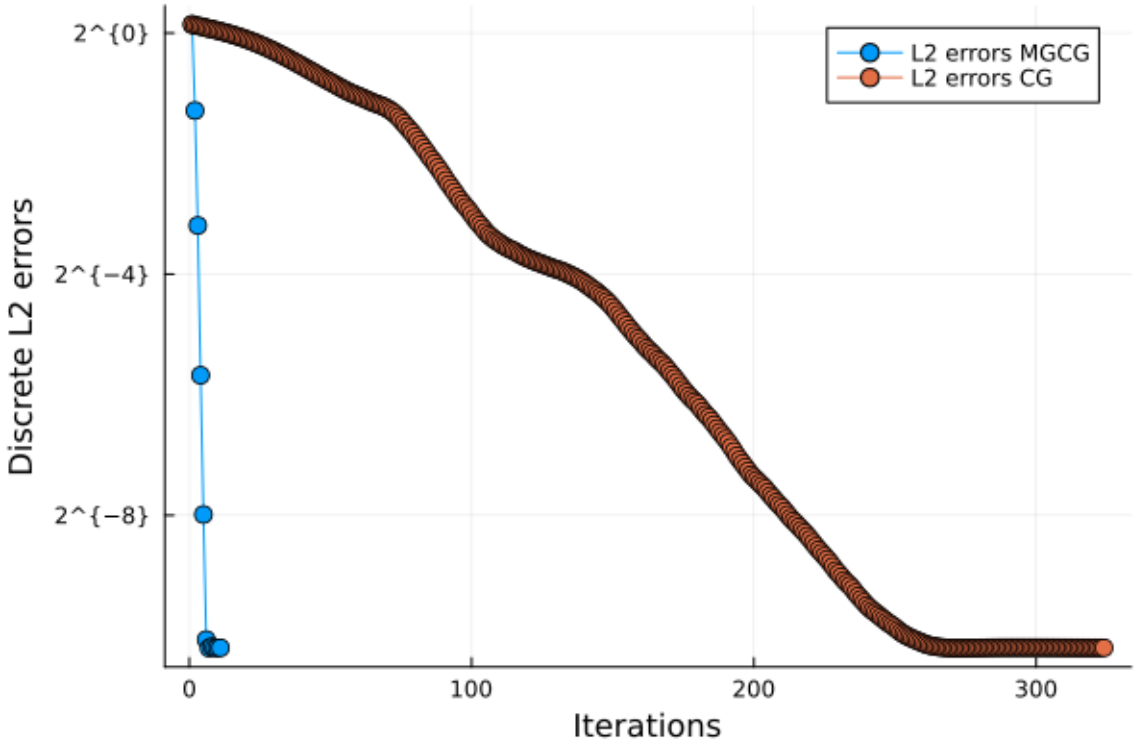

Figure 3. Log of error (difference from a direct solve) versus iteration count for multigrid preconditioned conjugate gradient (MGCG), shown in blue circles, using 5 steps of pre- and post-Richardson smoothing for every level versus unpreconditioned conjugate gradient (CG), shown in orange circles, for $N = 2^5$.

2.7 Parallel Processing and HPC

2.7.1 Parallel Implementation of Iterative Methods.

The iterative methods are ideal for their low memory requirements, and this becomes extremely important as the simulations in many fields of study have moved towards three-dimensional models. Another appealing part of iterative methods is that they are far easier to implement in parallel than sparse direct methods because they only require a small set of computational kernels. However, iterative methods are usually slower than direct methods, especially for smaller problems, requiring suitable preconditioning techniques for accelerated convergence. The parallel aspect of preconditioners also becomes very important naturally.

This subsection gives a short overview of various parallel architectures as well as different types of operations in iterative methods that can be parallelized.

There are currently three leading architectures of parallel models around which modern parallel computers are designed. These are

- The shared-memory model
- Single-instruction-multiple-data (SIMD)
- The distributed memory message passing model

2.7.1.1 Shared memory computers. A shared memory computer has processors connected to a large global memory, and the address space is the same for all processors. Data stored in a large global memory is readily accessible to any processor. There are two possible implementations of shared memory machines:

- bus-based architectures
- switch-based architectures

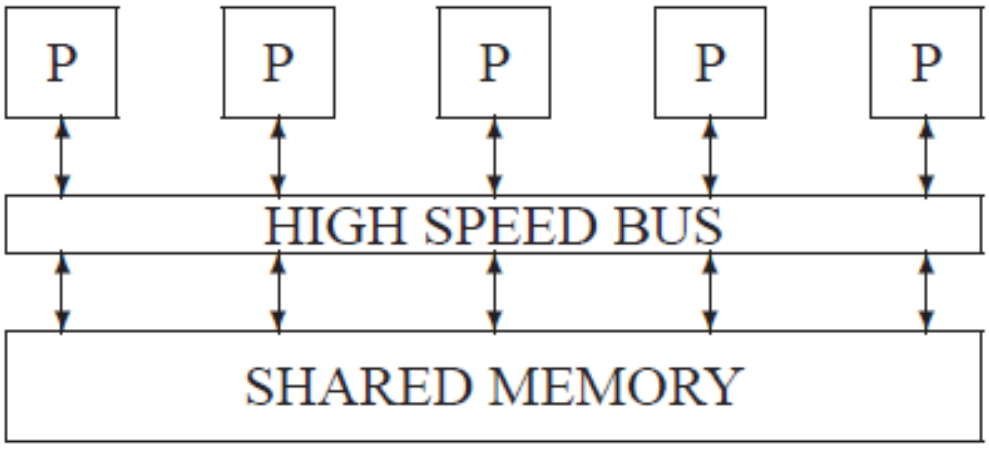

Figure 4. A bus-based shared memory computer Saad (2003)

These two architectures are illustrated in Figure 4 and Figure 5. So far, the bus-based model has been used more often. Buses are the backbone for communication between the different units of most computers, and usually have higher bandwidth for data I/O. The main reason why the bus-based model is more common is that the hardware involved in such implementation is simpler ADELI and VISHNUBHOTLA (1987). However, memory conflicts as well as the necessity

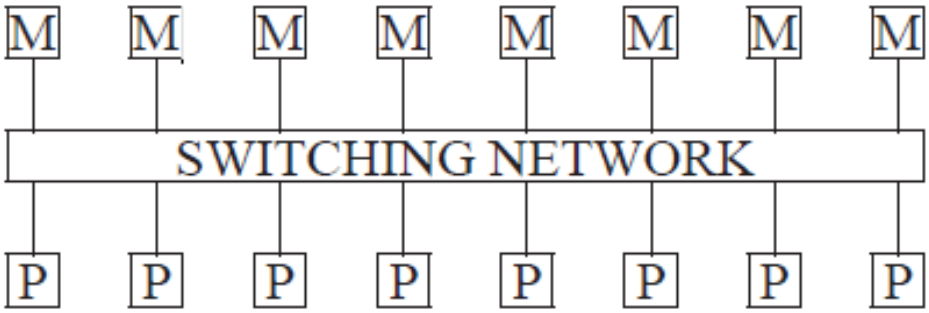

Figure 5. A switch-based shared memory machine Saad (2003)

to maintain data coherence can lead to worse performance. Moreover, shared memory computers can not take advantage of data locality in problems such as solving PDEs. Some machines can even have logically shared but physically distributed memory.

2.7.1.2 Distributed Memory Architectures. The *distributed memory* model can refer to distributed memory SIMD architecture or distributed memory with *memory passing* interface. A typical distributed memory system consists of a large number of identical processors and each processor has its own memory. These processors are interconnected in a regular topology. This can be shown with Figure 6. In these diagrams, each processor unit can be viewed as a complete processor with its one memory, CPU, I/O subsystems, control unit, etc. These processors are linked to a number of "neighboring" processors. In the "message passing" model, no global synchronizations are performed of the parallel tasks. Instead, computations are *data driven* because each processor performs a given task only when the operands it requires become available. The programmer needs to explicitly instruct data exchanges between different processors.

In the SIMD model, a different approach is used. A host processor stores the program and each slave processor holds different data. The host broadcasts instructions to each processor to execute them simultaneously. One advantage of this approach is that there is no need for large memories in each node to store the main program since the same instructions are broadcast to each processor.

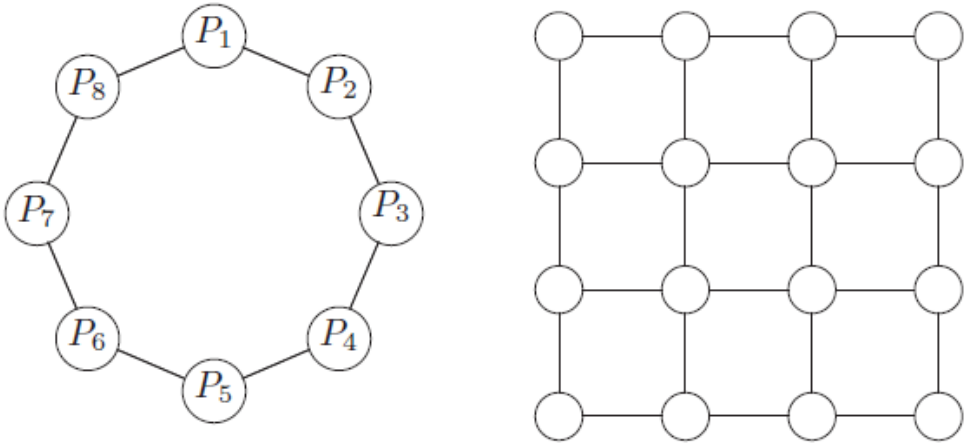

Figure 6. An eight-processing ring (left) and 4 × 4 multiprocessor mesh (right) Saad (2003)

Unlike the shared-memory model, distributed memory computers can easily exploit the data locality of data to reduce communication costs. Modern GPUs are designed with the SIMD model (more accelerately Single-instruction-multiple-threads, SIMT) Owens et al. (2008), and clusters with multiple CPUs are connected using a message passing interface (MPI) Barker (2015).

2.7.2 Key Operations in Parallel Implementation.

2.7.2.1 Types of Operations. We use the preconditioned Conjugate Gradient to demonstrate the typical operations involved that can be parallelized. The PCG algorithm consists of the following types of operations:

- Preconditioner setup
- Matrix-vector multiplications
- Vector update
- Dot Product
- Preconditioning operations

Matrix-vector multiplications, vector updates, and dot products are common operations in so-called Basic Linear Algebra Subprograms (BLAS) that can be easily parallelized and have been well studied and implemented for dense matricesChtchelkanova, Gunnels, Morrow, Overfelt, and Van De Geijn (1997); Dongarra, Du Croz, Hammarling, and Duff (1990); Freeman and Phillips

(1992). In terms of the computational costs, the vector update and dot product are much lower compared to the matrix-vector multiplication, which can still be carried out very quickly on the latest GPUs. The tricky part or the bottleneck for both memory and the runtime lies in the first step of preconditioner setup and the last step of preconditioning operations. We will discuss these two key operations in the next subsection. For now, let's focus on the Matrix-vector multiplication that has much higher computational costs than the vector update and the dot product.

2.7.2.2 ***Sparse Matrix-vector Products.*** The linear system coming from the discretization in the finite difference method is often sparse, which allows us to store them efficiently and use sparse matrix-vector products (SpMVs) for efficient computation. Different formats for storing sparse matrices can be found in Saad (2003). Compressed Sparse Row (CSR) sparse matrix format is one of the earliest sparse formats developed. It is ideal for parallelization since the data from each row can be handled independently. The SpMV algorithm for CSR format as well as the demonstration of the storage scheme of the CSR format is shown in Figure 7

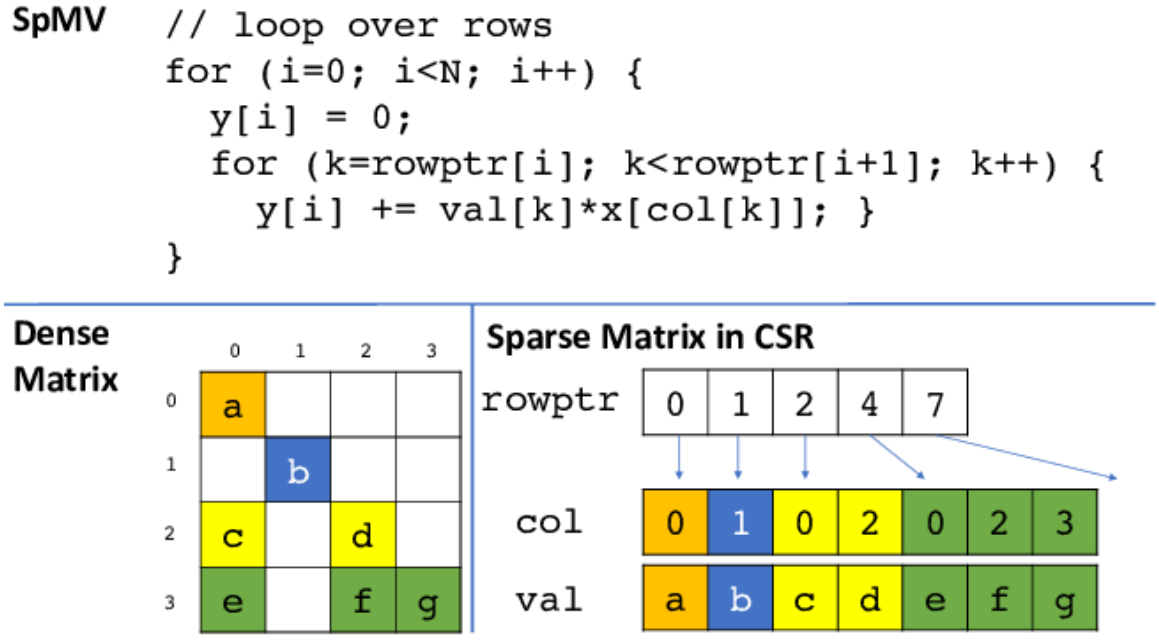

Figure 7. The `val` array stores the nonzeros by packing each row in contiguous locations. The `rowptr` array points to the start of each row in the `val` array. The `col` array is parallel to `val` and maps each nonzero to the corresponding column.Mohammadi et al. (2018)

A summary of different sparse matrix formats in the following table as well as a detailed performance comparison of these different formats can be found in Stanimirovic and Tasic (2009).

There are also recent work on developing new sparse matrix formats for optimal performance for

Short	Name	Short	Name
DNS	Dense	Ell	Ell-pack ItPack
BND	Linpack Banded	DIA	Diagonal
COO	Coordinate	BSR	Block Sparse Row
CSR	Compressed Sparse Row	SSK	Symmetric Skyline
CSC	Compressed Sparse column	BSR	Nonsymmetric Skyline
MSR	Modified CSR	JAD	Jagged Diagonal
LIL	Linked List		

Table 1. A summary of different sparse matrix formats and their short names

different use cases Dongarraxz, Lumsdaine, Niu, Pozoz, and Remingtonx (1994); Smailbegovic, Gaydadjiev, and Vassiliadis (2005) or implementing SpMV algorithms on parallel architectures Bell and Garland (2009); Li, Yang, and Li (2014); Yan, Li, Zhang, and Zhou (2014). Using the right sparse matrix formats and implementing them on suitable architectures can reduce the time spent on these SpMV calculations significantly, which makes iterative methods run faster. Another way to accelerate these iterative methods is to use the preconditioners that we mentioned before. A parallel implementation of these preconditioners becomes more challenging because of the complex arithmetic operations compared to the SpMV or other BLAS operations. We will focus on the parallel preconditioning technique in the next subsection.

2.7.3 Parallel Preconditioning.

2.7.3.1 Parallelism in Solving Linear Systems. Each preconditioned step from the previous subsection requires the solution of a linear system of equations of the form $Mz = y$. We consider traditional preconditioners such as ILU or SOR or SSOR, in which a solution with M is the result of solving triangular systems. Since these preconditioners are commonly used, it's important to explore their efficient parallel implementations for the iterative methods to be parallel. These preconditioners are mostly implemented on shared memory parameters. The distributed memory computers would use different strategies. These preconditioners require some sort of factorization, and the parallelism is done by sweeping through the lower triangular matrix or upper triangular matrix. Typical parallelism can be seen using a forward sweep.

It's typical for solving a lower triangular system that the solution is overwritten onto the right-hand side. So there is only one array u needed for both the solution and the right-hand side.

The forward sweep for solving lower triangular systems with coefficients $A(i,j)$ and right-hand-side b is defined as follows:

Algorithm 2 Sparse Forward Elimination

```
1: for i ← 2 to n do
2:     for each j such that A(i, j) ≠ 0 do
3:         u(i) ← u(i) − A(i, j) × u(j)
4:     end for
5: end for
```

Here $A(i,j)$ refers to the element in the sparse matrix. The different sparse matrix formats will have different implementations of locating this element, so the inner for loop will be implemented differently and the $A(i,j)$ will be replaced by different indexing code in different sparse matrix formats.

2.7.3.2 Parallel preconditioners. Several techniques can be for parallel implementations of the preconditioners. They can be summarized into three types of techniques. The simplest approach is to use a Jacobi or block Jacobi approach. In this case, a Jacobi preconditioner may be consist of a diagonal or a block-diagonal of A To improve the performance, these preconditioners can be accelerated by polynomial iterations. For example, the second level of preconditioning is called *polynomial preconditioning*.

A different strategy is to enhance parallelism by using graph algorithms, such as graph-coloring techniques. The gist of this approach is that all unknowns associated with the same color can be determined simultaneously in the forward or backward sweeps.

The third strategy uses generalizations of "partitioning" techniques which can be also called "domain decomposition" approaches.

We will give a brief overview of these methods in this part.

Overlapping block-Jacobi preconditioning is a parallel preconditioner similar to the general block-Jacobi approach with overlapping blocks as shown in Figure 8. Enlarging a system of algebraic equations by including duplicate copies of several rows, leads to an efficient iterative scheme on a multiprocessor MIMD array Wait and Brown (1988).

Polynomial preconditioners are another family of parallel preconditioners. In polynomial preconditioners, the matrix $\boldsymbol{M}$ is defined by $\boldsymbol{M^{-1} = s(A)}$, where $\boldsymbol{s}$ is a polynomial, typically of

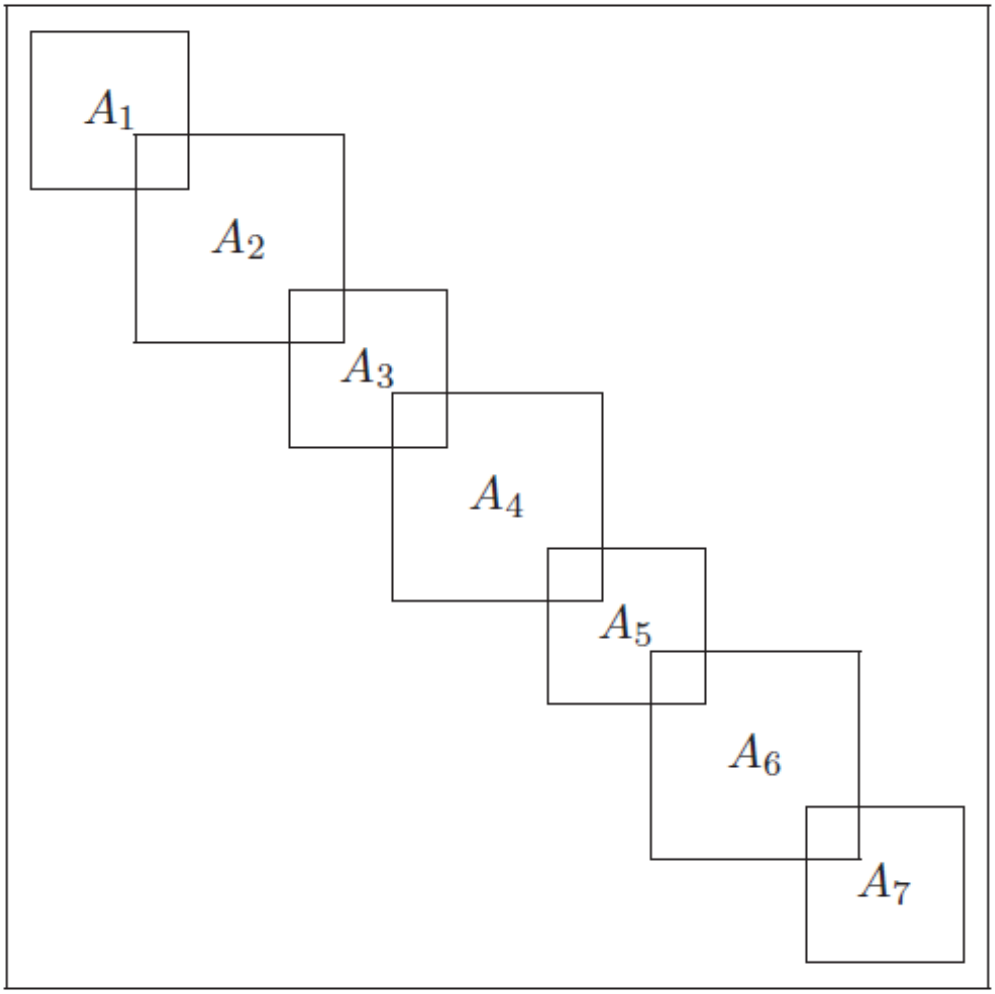

Figure 8. The block-Jacobi matrix with overlapping blocks Saad (2003)

low degree. Thus the original system can be preconditioned by

$$\boldsymbol{s}(\boldsymbol{A})\boldsymbol{A}\boldsymbol{u} = \boldsymbol{s}(\boldsymbol{A})\boldsymbol{f} \tag{2.113}$$

Note that the $\boldsymbol{s}(\boldsymbol{A})$ and $\boldsymbol{A}$ commute, and as a result, the preconditioned matrix is the same for left or right preconditioning. In addition, the matrix $\boldsymbol{s}(\boldsymbol{A})$ or $\boldsymbol{A}\boldsymbol{s}(\boldsymbol{A})$ does not need to be formed explicitly in matrix form, which allows the use of matrix-free methods. This approach was initially motivated by the good performance of matrix-vector operations on vector computers. It has now become more popular on iterative methods for GPU computing because of the similar SIMD architecture. There are several ways to construct polynomials in this method.

One of the most commonly studied approach is called the Chebyshev iteration that can be found in Saad (2003). One nice feature of the Chebyshev iteration is that it does not require inner products, and this is very attractive for parallel implementation as it does not require reductions.

Other polynomials include least-squares polynomials. A comparison of Chebyshev polynomials and least-square polynomials can be found in Ashby, Manteuffel, and Otto (1992). So far, Chebyshev polynomials have been the most popular for parallel implementation, especially in the matrix-free setting where the assembly of the matrix can be very expensive. Multicolor preconditioners are similar to ILU preconditioners in the sense that the construction and factorization of the matrices are required. Methods like these can be done in parallel, but they are not suitable for GPUs.

2.7.4 GPU architecture and CUDA. Given the increasing importance and popularity of GPUs in modern supercomputers, this subsection is dedicated to GPU architecture. As NVIDIA GPUs are mostly used in the industry for scientific computing and machine learning, the GPU programming model will be focused on CUDA (Compute Unified Device Architecture) toolkit.

A GPU is built as a scalable array of multithreaded *Streaming Multiprocessors* (SMs), each of which consists of multiple *Scalar Processor* (SP) cores. To manage hundreds or thousands of threads, the multiprocessors employ a *Single Instruction Multiple Threads* (SIMT) model with each thread mapped into one SP core and executing independently with its own instruction address and register state. Threads are organized in *warps*. A warp is defined as a group of 32 threads of consecutive thread IDs. More detailed information on optimizing memory access patterns can be found in Wilt (2013).

The NVIDIA GPU platform has various memory architectures. The types of memory can be classified as follows:

- off-chip global memory
- off-chip local memory
- on-chip shared memory
- read-only cached off-chip constant memory and texture memory
- registers

The effective bandwidth of each type of memory depends significantly on the access pattern. Global memory is relatively large but has a much higher latency. Using the right access pattern such as memory coalescing and avoiding bank conflicts will help achieve good memory bandwidth.

GPUs were initially designed for graphics-related calculations such as image rendering. General-purpose GPU programming on NVIDIA GPUs is supported by the NVIDIA CUDA toolkit. CUDA programs use similar syntax to C++. The main code on the host (CPU) would invoke a *kernel grid* that runs on the device (GPU). The same parallel kernel is executed by many threads. These threads are organized into thread blocks. Blocks and threads are the logical division of the GPU and are mapped to the actual SMs. Thread blocks are split into warps scheduled by SIMT units. All threads in the same block share the same shared memory and can be synchronized by a barrier. Threads in a warp execute one common instruction at a time. This is referred as warp-level synchronization Wilt (2013). It's most efficient when 32 threads of a warp follow the same execution path. Branch divergence in which threads within the same warp are executing different instructions often causes worse performance/

CUDA is only a lower-level tool for direct kernel programming. Libraries built on top of CUDA allow users to directly use code and kernels written for different tasks without manually programming and optimizing kernels themselves. Existing common CUDA libraries that supports GPU SpMV operation include CUDPP (CUDA Data Parallel Primitives)Harris, Owens, Sengupta, Zhang, and Davidson (2007), NVIDIA Cusp library Dalton et al. (2014), and the IBM SpMV library Baskaran and Bordawekar (2009). In these packages, different formats of sparse matrices are studied for producing high-performance SpMV kernels on GPUs. These include the compressed sparse row (CSR) format, the coordinate format (COO), the diagonal (DIA) format, the ELLPACK (ELL) format., and a hybrid (ELL/COO) format. There are other recent sparse matrix formats specifically designed for GPU computing, but we will not go into detail to cover each of them.

For dense linear algebra computations, the MAGMA (Matrix Algebra for GPU and Multicore Architectures) project hybrid multicore-multi-GPU system aims to develop a dense linear algebra similar to LAPACKAgullo et al. (2009). Since our numerical methods for PDEs would generate a sparse linear system, we did not explore this library in this paper.

CHAPTER III

SCIENTIFIC COMPUTING LIBRARIES AND LANGUAGES

3.1 PETSc

PETSc, which stands for Portable, Extensible Toolkit for Scientific Computation, is a software library developed primarily by Argonne National Library to facilitate the development of high-performance parallel numerical code written in C/C++, Fortran and Python. It provides a wide range of functionality for solving linear and nonlinear algebraic equations, ordinary and partial differential equations, and also optimization problems (provided by TAO) on parallel computing architectures. In addition, PETSc includes support for managing parallel PDE discretizations including parallel matrix and vector assembly routines.

Key features of PETSc include:

- Parallelism: PETSc is designed for parallel computing, especially distributed-memory parallel computing architectures. It is intended to run efficiently on parallel computing systems where multiple processors or nodes communicate over the network via a message passing interface (MPI). These architectures include clusters, supercomputers, and other HPC platforms.

- Modularity and Extensibility: PETSc is highly modular and extensible, allowing users to combine different numerical techniques and algorithms to solve complex problems efficiently. It provides a flexible framework for implementing new algorithms and incorporating external libraries. It mainly contains the following objects

    * Algebraic objects

        · Vectors (Vec) containers for simulation solutions, right-hand sides of linear systems

        · Matrices (Mat) containers for Jacobians and operators that define linear systems

    * Solvers

        · Linear solvers based on preconditioners (PC) and Krylov subspace methods (KSP)

        · Nonlinear solvers (SNES) that use data-structure-neutral implementations of Newton-like methods

        · Time integrators (TS) for ODE/PDE, explicit, implicit, IMEX

· Optimization (TAO) with equality and inequality constraints, first and second order Newton methods

· Eigenvalue/Eigenvectors (SLEPc) and related algorithms

– Efficiency and Performance: PETSc is optimized for performance, with algorithms and data structures designed to minimize memory usage and maximize computational efficiency. It supports parallel matrix and vector operations as well as efficient iterative solvers and preconditioners via the objects mentioned previously

– Flexibility: PETSc supports a wide range of numerical methods and algorithms and has built-in discretization tools. It provides interfaces for solving problems in various scientific and engineering disciplines, including computational fluid dynamics (CFD), solid mechanics, etc with documented examples and tutorials for researchers.

– PETSc is portable across different computing platforms and operating systems, including UNIX/Linux, macOS, and Windows. It provides a consistent interface and functionalities across different architectures, making it easy to develop and deploy simulation code across multiple platforms.

3.2 AmgX

AmgX is a proprietary software library developed by NVIDIA to accelerate the solution of large-scale linear systems arising from finite element and finite volume discretizations typically found in computational fluid dynamics (CFD) and computational mechanics simulations on NVIDIA GPUs. AmgX stands for Algebraic Multigrid Accelerated. It provides wrappers to work with other libraries like PETSc and programming languages like Julia.

Key features of AMGX include:

– Preconditioning: AmgX offers a variety of advanced preconditioning techniques, including algebraic multigrid (AMG), smoothed aggregation and hybrid methods to accelerate the convergence of iterative solvers for sparse linear systems. These preconditioners are designed for and tested in real-world engineering problems in collaboration with companies like ANSYS, a provider of leading CFD software Fluent.

– Parallelism: AmgX is optimized for NVIDIA GPUs and provides support for OpenMP to allow acceleration via heterogeneous computing and MPI to run large simulations across multiple GPUs and clusters.

– Flexibility and Customization: AmgX offers a flexible and extensible framework for configuring and customizing the solver algorithms via JSON files.

The limitation of AmgX is due to its link with NVIDIA. It can not run on GPUs from other vendors, such as AMD and Intel. Some of the latest exascale supercomputers are built with CPUs and GPUs from AMD and Intel.

3.3 HYPRE

HYPRE is a software library of high performance numerical algorithms including preconditioners and solvers for large, sparse linear systems of equations on massively parallel computers Falgout, Jones, and Yang (2006b). The HPYRE library was created to provide users with advanced parallel preconditioners. It features parallel multigrid solvers for both structured and unstructured grid problems. These solvers are called from application code via HYPRE's conceptual linear system interfaces Falgout, Jones, and Yang (2006a), which allow a variety of natural problem descriptions.

Key features of HYPRE include:

– Scalable preconditioners: HYPRE contains several families of preconditioners focused on scalable solutions of very large linear systems. HYPRE includes "grey box" algorithms including structured multigrid that use more than just the matrix to solve certain classes of problems more efficiently.

– Common iterative methods: HYPRE provides several common Krylov-based iterative methods in conjunction with scalable preconditioners. This includes methods for symmetric matrices such as Conjugate Gradient (CG) and nonsymmetric matrices such as GMRES.

– Grid-centric interfaces for complicated data structures and advanced solvers: HYPRE has improved usability from earlier generations of sparse linear solver libraries in that users do not have to learn complicated sparse matrix data structures. HYPRE builds these data structures for users through a variety of conceptual interfaces for different classes of users. These include stencil-based structured/semi-structured interfaces most suitable for finite

difference methods, unstructured interfaces for finite element methods, and linear algebra based interfaces for general applications. Each conceptual interface provides access to several solvers without the need to manually write code for new interfaces.

- User options for beginners through experts: HYPRE allows users with various levels of expertise to write their code easily. The beginner users can set up runnable code with a minimal amount of effort. Expert users can take further control of the solution process through various parameters

- Configuration options for different platforms: HYPRE allows a simple and flexible installation on various computing platforms. Users have options to configure for different platforms during the installation. Additional options include debug mode which offers more info and optimized mode for better performance. It also allows users to change different libraries such as MPI and BLAS.

- Interfaces to multiple languages: HYPRE is written in C, but it also provides an interface for Fortran users.

3.4 Review of several languages for scientific computing

3.4.1 Fortran. There are many languages designed for high performance computing. Traditionally, Fortran has been used to write high performance numerical code. It is short for "Formula Translation". As the name suggest, it is one of the oldest and most enduring programming languages in scientific computing. Developed in the 1950s by IBM, it was designed to facilitate numerical and scientific computations, particularly for high-performance computing on mainframe computers.

Fortran was specifically designed for efficient numerical and scientific computing, with optimized operations handling mathematical operations, arrays, and complex computations. It provides a rich set of built-in functions and libraries for numerical analysis, linear algebra, differential equations, and other mathematical tasks. It is a statically typed language, meaning that variable types are declared explicitly at compile time and do not change during runtime. This allows compilers to perform extensive type checking and optimization to generate efficient code for execution.

Fortran codes are also highly portable across different computing platforms. While early versions of Fortran (66, 77) were designed for specific hardware architectures, modern Fortran standards, such as Fortran 90, 95, 2003, 2008, and 2018 (formerly 2015) have introduced features that enhance portability and interoperability with other languages and systems. Fortran is also known for its excellent backward compatibility, with newer language standards preserving compatibility with older databases. This allows legacy Fortran programs to continue running without modification on modern compilers and systems, ensuring long-term viability and support for existing applications, which is very important in scientific research where many simulation codes are built on top of decades of previous work.

Because of these reasons, despite its age, Fortran remains widely used in scientific and engineering computing.

3.4.2 C and C++. C was created in 1972 as a general-purpose programming language. C++ was created in 1979 to enhance C language with object-oriented design and many useful standard template libraries. Despite the historical dominance of Fortran in scientific and engineering computing, C and C++ have gradually replaced Fortran in many scientific computing and HPC codes due to their performance, flexibility, and rich ecosystem of tools and libraries.

While Fortran continues to be used in certain domains, particularly in legacy codebases and specialized applications, the adoption of C and C++ as the default language in many modern packages reflects the evolving needs and preferences of HPC developers for modern, versatile programming languages.

C and C++ are known for their performance and efficiency. In fact, they are often used as the standard to compare the performance of various programming languages. This is because they provide low-level control over hardware resources and memory management, allowing programmers to write code that executes with high speed and minimal overhead. The performance is crucial for HPC applications, which often involve computationally extensive tasks and large-scale simulations. Known as high-level languages, C and C++ strike a balance between high-level abstractions and low-level control. They support multiple programming paradigms including procedural, and object-oriented. C/C++ can also be extended to handle parallel processing via pragma directives. This allows the creation of modular, reusable code with

encapsulation, inheritance, polymorphism, and templates. Standard Libraries built on top of these features provide implementations of fundamental data structures, algorithms, and utilities.

In addition to their language features, C and C++ offer support for concurrency and parallelism via low-level features like threads, mutexes, condition variables, atomic operations, and parallel algorithms. Modern C++ standards (such as C++11, C++17 and C++20) have introduced high-level features to manage asynchronous execution, parallel computation, and parallelism-aware data structures. All these efforts further enhance the capability of C and C++ as high performance computing languages.

As general-purpose programming languages, C and C++ codes are highly portable across different platforms and architectures. The portability is essential for deploying HPC applications on diverse computing platforms, including cloud servers, clusters, and supercomputers. C and C++ also have excellent interoperability with other programming languages and systems. They can be easily integrated with libraries and tools including most common HPC languages like Fortran, Python, and CUDA. This interoperability allows developers to leverage existing software components and take advantage of specialized and optimized libraries for specific computational tasks. However, the impact on the performance needs to be considered carefully when interoperating C and C++ with other languages.

3.4.3 MATLAB. MATLAB is a high-level programming language usually used in an interactive development environment (IDE) from the software with the same name. Developed by MathWorks, it is widely used for numerical computing, data analysis, visualization, and algorithm development. Compared to compiled languages that can generate binary executables running natively on operation systems, MATLAB requires an interpreter (usually by MATLAB) to "translate" the code whenever the code is run. To avoid ambiguity, we refer to both the language and the IDE as MATLAB here. As a proprietary language and tool, MATLAB offers limited access to the source code, and it is prohibitively expensive for people outside of academia without an educational license. GNU Octave is used as an open source alternative to MATLAB as it is mostly compatible with MATLAB. Octave is free and lightweight, however, it often comes with the cost of worse performance. Despite being a proprietary software, MATLAB is still often used in scientific computing, especially in academia for the following reasons:

MATLAB is easy to use because of its intuitive syntax for mathematicians and comprehensive set of built-in functions for numerical computing, including matrix manipulation, linear algebra, and optimizations. For these functions, MATLAB offers extensive examples and tutorials, making it a great choice for beginners for learning and advanced users for writing code.

MATLAB has an interactive environment with visualization tools that enable users to iterate quickly on algorithms. It offers a command-line interface that is similar to read-evaluate-print-loop (REPL) in interpreted languages like Python, and also integrates many common functionalities via UI buttons in its IDE. Like many IDEs, MATLAB provides tools for organizing code, debugging, profiling, and version control. More importantly, MATLAB's functionality can be extended through its proprietary and third-party toolboxes, which are collections of specialized functions and algorithms for specific domains of applications such as signal processing, control theory, and statistics.

Because of these features and accessibility via academic licenses through educational institutes, many people start numerical coding in MATLAB and continue to develop in MATLAB for research purposes. Although MATLAB is designed to run numerical calculations efficiently and also provides some limited support for parallel and GPU computing, it was not designed as a HPC language running on clusters, supercomputers, and cloud infrastructures. Researchers often use MATLAB for quick implementation and testing during the prototyping stage and then rewrite their code in HPC languages such as FORTRAN and C/C++. This raises the so-called "two-language" problem which inspires the development of the Julia language.

3.5 Julia langauge

Julia is a dynamically typed language for scientific computing designed with high performance in mind Bezanson, Edelman, Karpinski, and Shah (2017). Julia supports general-purpose GPU computing with the package CUDA.jl. Through communications in LLVM intermediate representations with NVIDIA's compiler, it is claimed that CUDA.jl achieves the same level of performance as CUDA C according to previous researchBesard, Foket, and De Sutter (2018). Aimed to address the "two-language" problem, Julia enables implementation ease of complex mathematical algorithms while achieving high performance, an ideal match for computational scientists without expertise in low-level language-based HPC. Julia has gained attention among the HPC community, with notable examples including The Climate Machine

Sridhar et al. (2022), a new Earth System model written purely in Julia that is capable of running on both CPUs and GPUs by utilizing KernelAbstractions.jl Churavy et al. (2024), a pure Julia device abstraction similar to Raja, Kokkos, and OCCA Beckingsale et al. (2019); Carter Edwards, Trott, and Sunderland (2014); Medina, DS and St-Cyr, A. and Warburton, T (2014). In addition, because a large body of researchers studying SBP methods use Julia in serial, e.g. Kozdon et al. (2020); Ranocha and Nordström (2021), our developments will enable these users to gain HPC capabilities in their code with minimal overhead.

CHAPTER IV

MATRIX-FREE SBP-SAT METHODS ON GPUS

4.1 Partial Differential Equations for the Solid Earth

4.1.1 Governing equations and boundary conditions. Static deformation of the solid Earth over the time-scales of earthquake cycles is governed by the equilibrium equation and a constitutive relationship describing the material properties. The standard assumption is that the Earth is linear elastic, defined on a sub-domain of $\mathbb{R}^3$. While solutions to the 3D elasticity equations are the eventual target, 2D problems are considered in this work in order to sort out implementation details with reduced computational costs. The 2D anti-plane shear problem Lubarda and Lubarda (2019) is particularly ubiquitous in earthquake applications, where a vertical cross-section of a 3D problem (assuming invariance in one-direction) gives rise to an elliptic equation, where only one non-zero component of the displacement vector exists and depends on two spatial variables, namely,

$$-\nabla \cdot (\mu \nabla u) = f \quad \text{for } (x, y) \in \Omega, \tag{4.1}$$

where $\mu(x, y)$ is the spatially-varying shear modulus, u is Earth's material displacement in the z-direction, and f comprises body forces. In order to enable complex fault geometries and topography, we assume that Ω is a curved quadrilateral in $\mathbb{R}^2$, which enables extensions to arbitrary domains partitioned into computational blocks, (e.g. Kozdon et al., 2020). As **??**(a) illustrates, the boundary can be partitioned into four curves $\partial\Omega_i,\ i = 1, ..., 4$, where (for example), $\partial\Omega_3$ represents Earth's surface. The shear modulus $\mu(x, y)$ can vary in order to represent heterogeneities in the crust, for example, a sedimentary basin, which is known to trap waves, extend shaking, and increase earthquake magnitudes, (e.g. Boué, Denolle, Hirata, Nakagawa, & Beroza, 2016).

In this work both Dirichlet and Neumann boundary conditions are considered for generality, as each appears in earthquake problems. For example Earth's free surface manifests as a Neumann condition and slow motion of tectonic plates is usually enforced via a Dirichlet condition B. A. Erickson and Dunham (2014). As proof of concept, we consider boundary

conditions

$$u = g_1, \quad \partial\Omega_1, \tag{4.2a}$$

$$u = g_2, \quad \partial\Omega_2, \tag{4.2b}$$

$$\boldsymbol{n} \cdot \mu\nabla u = g_3, \quad \partial\Omega_3, \tag{4.2c}$$

$$\boldsymbol{n} \cdot \mu\nabla u = g_4, \quad \partial\Omega_4, \tag{4.2d}$$

where vector $\boldsymbol{n}$ is the outward pointing normal to the domain boundary and $g_i,\ i\ =\ 1, ..., 4$ represent boundary data.

4.1.2 Coordinate Transformation. In order to solve Equation 4.1-Equation 4.2 with SBP-SAT methods, the domain Ω is transformed (via a conformal mapping) to the regular, square domain $(r, s) \in \bar{\Omega} = -1 \leq (r, s) \leq 1$, as in **??**(b). The transformed equations are given by

$$-\bar{\nabla} \cdot \left(\mathbf{c}\bar{\nabla}u\right) = Jf, \qquad \text{for } (r, s) \in \bar{\Omega}, \tag{4.3a}$$

$$u = g_1, \qquad \text{face 1}, \tag{4.3b}$$

$$u = g_2, \qquad \text{face 2}, \tag{4.3c}$$

$$\hat{\boldsymbol{n}}^3 \cdot \mathbf{c}\bar{\nabla}u = S_J^3 g_3, \quad \text{face 3}, \tag{4.3d}$$

$$\hat{\boldsymbol{n}}^4 \cdot \mathbf{c}\bar{\nabla}u = S_J^4 g_4, \quad \text{face 4}, \tag{4.3e}$$

where $\bar{\nabla}u = \left[\frac{\partial u}{\partial r}, \frac{\partial u}{\partial s}\right]^T$, face k for $k = 1, ...4$ define the domain boundaries of $\bar{\Omega}$, given in **??**(b). $J > 0$ is the Jacobian and S_J^k is the surface Jacobian on face k. Vector $\hat{\boldsymbol{n}}^k$ is the outward pointing normal to the face and 2×2 matrix $\mathbf{c}$ is symmetric positive definite (SPD) and encodes the variable material properties $\mu(x, y)$ and the coordinate transform, see B. A. Erickson et al. (2022); Kozdon et al. (2020) for more details.

4.2 Problem Discretization

The SBP-SAT discretization to Equation 4.3 is given by

$$-\tilde{\boldsymbol{D}}_{ij}^{(c_{ij})}\boldsymbol{u} = \boldsymbol{J}\boldsymbol{f} + \sum_k \boldsymbol{p}_k \tag{4.4}$$

where $\boldsymbol{p}_k$ are SAT vectors formed from the boundary condition on face k. To illustrate the structure of the SAT vectors, enforcing Dirichlet conditions on faces $k = 1, 2$ generates

$$\boldsymbol{p}_k = \left(\boldsymbol{H} \otimes \boldsymbol{H}\right)^{-1} \left(\boldsymbol{G}_k^T - \boldsymbol{L}_k^T \boldsymbol{H}_k \boldsymbol{\tau}_k\right) \left(\boldsymbol{L}_k \boldsymbol{u} - \boldsymbol{g}_k\right), \tag{4.5}$$

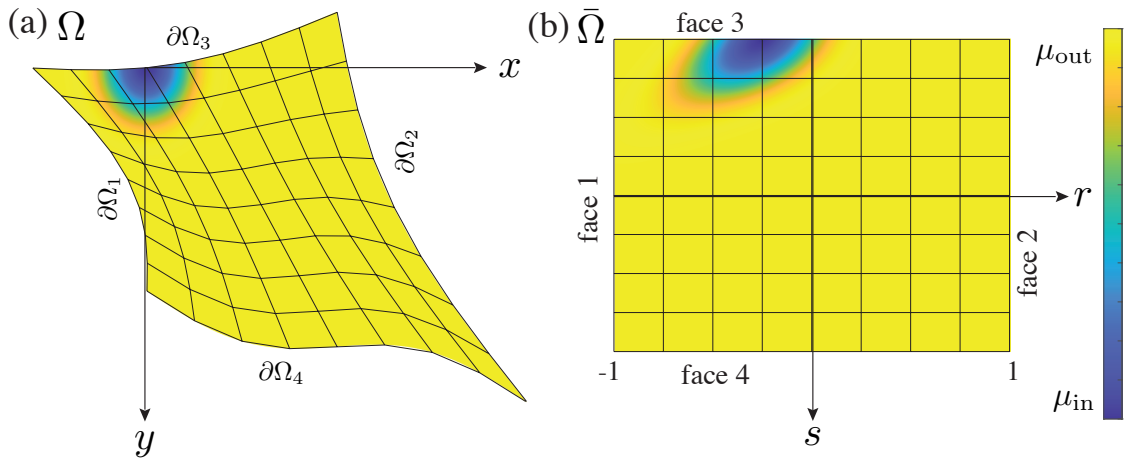

Figure 9. (a) Geometrically complex physical domain Ω with material stiffness that increases from μ_{in} within a shallow, ellipsoidal sedimentary basin, to stiffer host rock given by μ_{out}. (b) Ω is transformed to the regular, square domain $\bar{\Omega}$ via conformal mapping.

where matrix $\mathbf{G}_k = \hat{n}_i^k \boldsymbol{H}_k \boldsymbol{C}_{ij}^k \boldsymbol{B}_j^k$ computes the weighted derivative on face k. Matrix $\boldsymbol{\tau}_k = \hat{n}_i^k \boldsymbol{C}_{ij}^k \hat{n}_j^k \boldsymbol{\Gamma}_k$, where $\boldsymbol{\Gamma}_k$ is the diagonal penalty matrix on face k with sufficiently large components to ensure discrete stability, according to

$$\boldsymbol{\Gamma}_k \geq \frac{4}{h_\perp^k}\mathbf{I} + \frac{1}{h_\perp^k}\mathbf{P}_k, \tag{4.6}$$

where

$$\mathbf{P}_k = \begin{cases} \boldsymbol{C}_{rr}^k (\boldsymbol{C}_{rr}^{k,min})^{-1}, & \text{k} = 1,\, 2, \\ \boldsymbol{C}_{ss}^k (\boldsymbol{C}_{ss}^{k,min})^{-1}, & \text{k} = 3,\, 4. \end{cases} \tag{4.7}$$

Here $\boldsymbol{C}^{k,min}$ is the minimum value of c in the two points orthogonal to the boundary and $h_\perp^k$ is the grid spacing orthogonal to face k B. A. Erickson et al. (2022).

Imposing Neumann conditions on faces $k = 3, 4$ corresponds to SAT vectors

$$\boldsymbol{p}_k = -\left(\boldsymbol{H} \otimes \boldsymbol{H}\right)^{-1} \boldsymbol{L}_k^T \left(\boldsymbol{G}_k \boldsymbol{u} - \boldsymbol{H}_k \boldsymbol{S}_j^k \boldsymbol{g}_k\right). \tag{4.8}$$

In order to render the linear system Equation 4.4 SPD, we multiply by $(\boldsymbol{H} \otimes \boldsymbol{H})$ (the discrete equivalent of integrating over $\bar{\Omega}$ and discretizing the weak form). This process yields the final linear system

$$\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b} \tag{4.9}$$

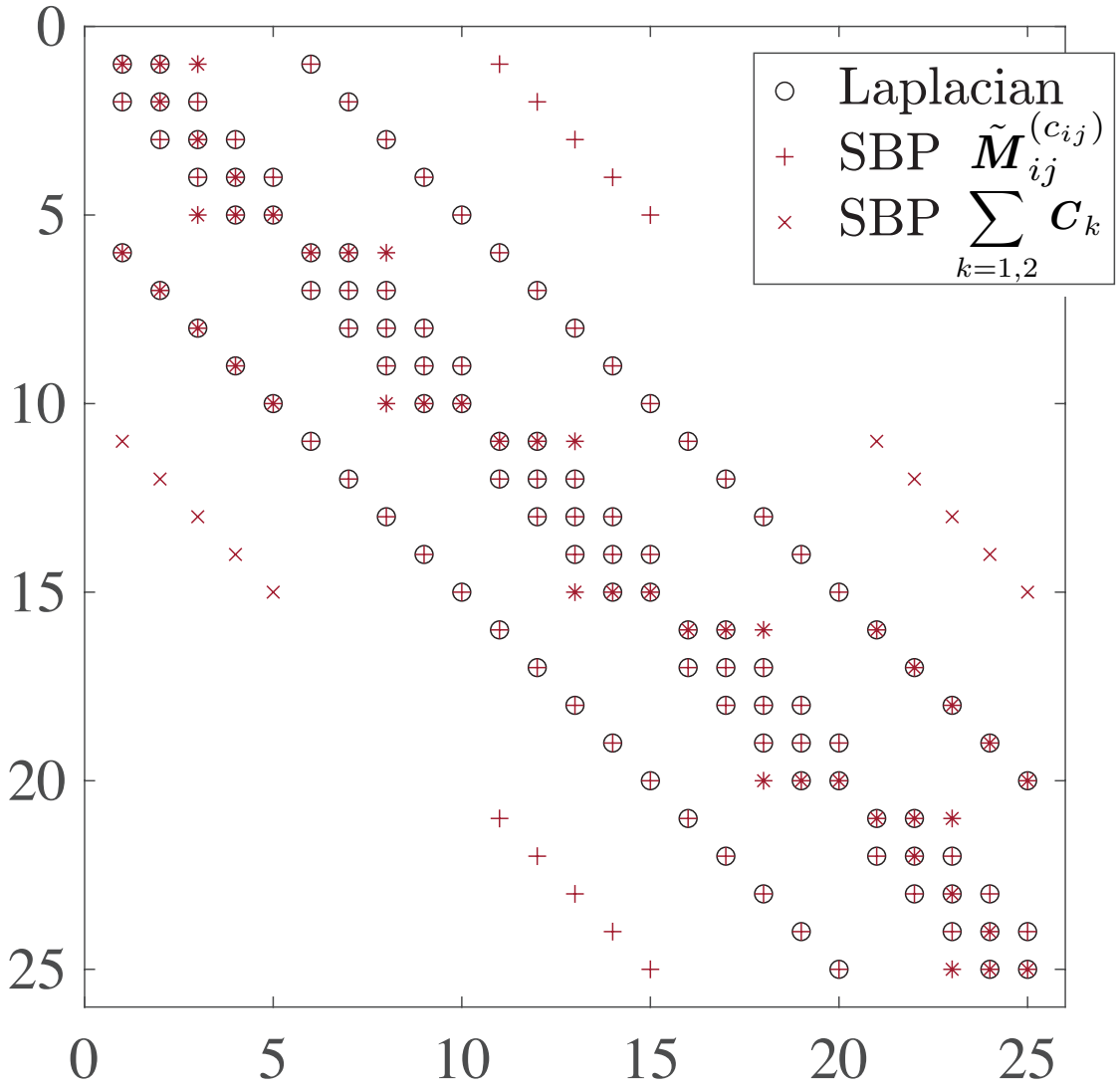

Figure 10. Sparsity pattern for matrix $\boldsymbol{A}$ with $N = 5$ grid points in each direction. Traditional 5-point Laplacian stencil in black circles. Contributions to $\boldsymbol{A}$ are separated into contributions from the volume (red $+$) and from the boundary enforcement (red $\times$), so that contributions from both (red $*$) cancel any deviations from symmetry.

where

$$\boldsymbol{A} = \tilde{\boldsymbol{M}}_{ij}^{(c_{ij})} + \sum_{k=1,2} \boldsymbol{C}_k, \tag{4.10}$$

is SPD B. A. Erickson et al. (2022), and matrices

$$\boldsymbol{C}_k = -\mathbf{L}_k^T \mathbf{G}_k - \mathbf{G}_k^T \mathbf{L}_k + \mathbf{L}_k^T \boldsymbol{H}_k \boldsymbol{\tau}_k \mathbf{L}_k. \tag{4.11}$$

Right-hand side vector $\boldsymbol{b}$ is given by

$$\boldsymbol{b} = (\boldsymbol{H} \otimes \boldsymbol{H}) \left[\boldsymbol{J}\boldsymbol{f} + \sum_k \boldsymbol{K}_k \boldsymbol{g}_k \right] \tag{4.12}$$

which encodes the source term and boundary data. Here matrices $\boldsymbol{K}$ depend on boundary conditions; for those given in Equation 4.3 they are

$$\boldsymbol{K}_1 = \boldsymbol{L}_1^T \boldsymbol{H}_1 \boldsymbol{\tau}_1 - \boldsymbol{G}_1^T \tag{4.13}$$

$$\boldsymbol{K}_2 = \boldsymbol{L}_2^T \boldsymbol{H}_2 \boldsymbol{\tau}_2 - \boldsymbol{G}_2^T \tag{4.14}$$

$$\boldsymbol{K}_3 = \boldsymbol{L}_3^T \boldsymbol{H}_3 \tag{4.15}$$

$$\boldsymbol{K}_4 = \boldsymbol{L}_4^T \boldsymbol{H}_4. \tag{4.16}$$

Note that $\boldsymbol{A}$ includes contributions from both volume operators ($\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$) and from the SAT enforcement of boundary terms ($\boldsymbol{C}_k$), and differs from the traditional discrete Laplacian near all domain boundaries; see Figure 10. At Dirichlet boundaries (faces 1 and 2), $\boldsymbol{C}_k$ modifies the layer of three points normal to the face (i.e. the SAT imposition penalizes all points used in the computation of the derivative normal to the face).

4.3 Matrix-free GPU kernels

In this chapterr, we develop custom, matrix-free GPU kernels (specifically for SBP-SAT methods) for computations in the volume and boundaries, which show improved performance as compared to the native, matrix-explicit implementation while requiring only a fraction of memory. GPU-acceleration of our resulting matrix-free, preconditioned iterative scheme shows superior performance compared to state-of-the-art methods offered by NVIDIA.

Stencil computations have proven efficient in utilizing GPU resources to achieve optimal performance Krotkiewski and Dabrowski (2013); Vizitiu, Itu, Niţă, and Suciu (2014). In this work we implement a similar GPU kernel for our 2D problem by matching each spatial node to a GPU thread, however, our work requires specialized treatment for domain boundaries. The most computationally expensive operator is the volume operator $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$, which differs from

traditional finite difference operators in that it involves derivative approximations at domain boundaries. However, the use of else statements in GPU kernels tends to lead to warp divergence and should be avoided. We construct the matrix-free action of $\boldsymbol{A}$, referred to as `mfA!`() based on node location. Kernel 3 provides the partial pseudocode, i.e. it includes pseudocode for the $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$ calculation; boundary condition calculations are further detailed in code block 1. At interior nodes the action of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$ is defined by a single stencil (with spatially varying coefficients). The action of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$ on boundary nodes, however, has a different stencil depending on the face number and whether the node is at a corner of the domain. To avoid race conditions at corners (while minimizing conditional statements), only normal components of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$ are computed (as they correspond to the same stencil). For example on face 1 only the action of $\tilde{\boldsymbol{M}}_{rr}^{c_{rr}}$ and $\tilde{\boldsymbol{M}}_{rs}^{c_{rs}}$ are computed at the corners, see Figure 11. The action of the remaining components of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$ on the corner nodes are computed in computations associated with adjacent faces (faces 3 and 4).

At boundary nodes we must also compute boundary condition operators $\boldsymbol{C}_k$, with differing stencils depending on face number and whether a node is an interior node, an interior boundary node (i.e. not a corner), or a corner node. Code block 1 provides the pseudocode for nodes on face 1; stencils are differentiated with superscripts int, sw, nw, corresponding to the interior boundary, northwest, and southwest corner nodes, respectively. Figure 11 further illustrates the nodes involved in each computation: black dots correspond to nodes within the 2D domain. Black circles correspond to the interior nodes that are modified by the action of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$. On the western boundary (face 1), the three-node layer adjacent to face 1 is used to compute the actions of the volume and boundary operators. Blue diamonds and red stars correspond to nodes that are modified by the different components of $\tilde{\boldsymbol{M}}_{ij}^{c_{ij}}$. Green squares correspond to the nodes that are modified by the boundary operator $\boldsymbol{C}_1$ in order to impose the Dirichlet condition (in this case a layer of three nodes normal to the face. More rows are involved for higher order p).

4.4 Performance: Matrix-free GPU kernels

4.4.1 Performance Comparison. With `mfA!`() we can carry out the matrix-vector product without explicitly storing the matrix. In this section, we compare its performance against the matrix-explicit cuSPARSE SpMV implementation available through CUDA.jl. We note that this is not an exhaustive comparison against all possible sparse matrix data structures. Our goal is to establish a baseline comparison of our matrix-free implementation against the

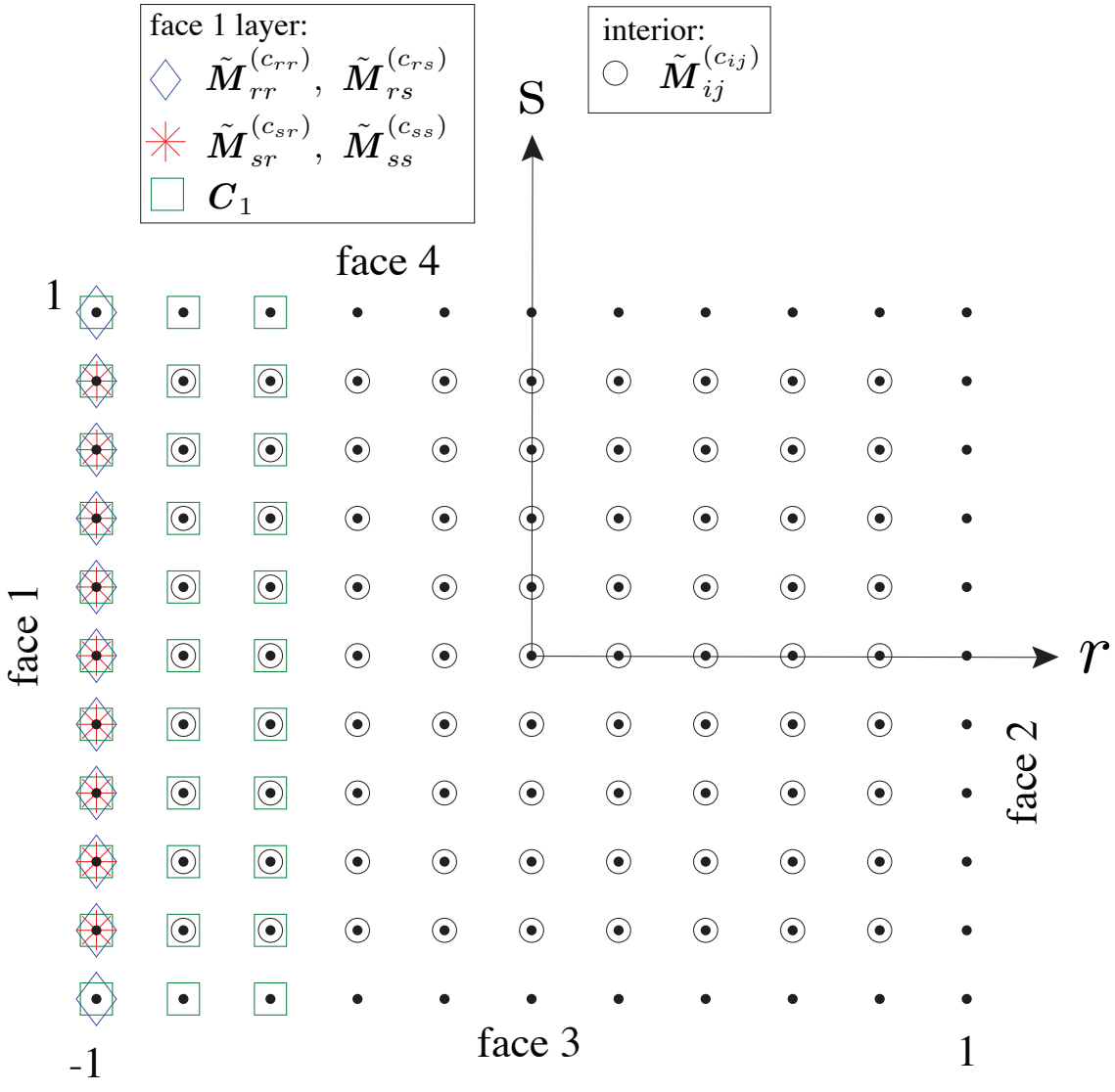

Figure 11. Schematic of 2D computational domain; nodes denoted with solid black dots. `mfA!`() modifies interior nodes, denoted with circles. For face 1, contributions to `mfA!`() from coordinate transformation matrices modify nodes corresponding to different shapes. Calculations by boundary operator $\boldsymbol{C}_1$ modify nodes in green squares.

Algorithm 3 Matrix-Free GPU kernel Action of matrix-free A for interior nodes.

function mfA!(odata, idata, $c_{rr}, c_{rs}, c_{ss}, h_r, h_s$)
i, j =get_global_thread_IDs()
$g = (i-1)*(N+1)+j$ ▷ compute global index
if $2 \le i, j \le N$ **then** ▷ interior nodes
odata[g] = (hs/hr)(- (0.5c_{rr}[g-1] + 0.5c_{rr}[g])idata[g-1] +
+ (0.5c_{rr}[g-1] + c_{rr}[g] - 0.5c_{rr}[g+1])idata[g] +
- (0.5c_{rr}[g] + 0.5c_{rr}[g+1])idata[g+1]) +
▷ compute M_{rr} stencil

+ 0.5c_{rs}[g-1](-0.5idata[g-N-2] + 0.5idata[g+N]) +
- 0.5c_{rs}[g+1](-0.5idata[g-N] + 0.5idata[g+N+1]) +
▷ compute M_{rs} stencil

+ 0.5c_{rs}[g-N-1](-0.5idata[g-N-2] + 0.5idata[g-N]) +
- 0.5c_{rs}[g+N+1](-0.5idata[g-N] + 0.5idata[g+N+2]) +
▷ compute M_{sr} stencil

- (0.5c_{ss}[g-N-1] + 0.5c_{ss}[g])idata[g-N-1] +
+ (0.5c_{ss}[g-N-1] + c_{ss}[g] + 0.5c_{ss}[g+N+1])idata[g] -
- (0.5c_{ss}[g] + 0.5c_{ss}[g+N+1])idata[g+N+1]))
▷ compute M_{ss} stencil
end if
... ▷ boundary nodes, e.g. Algorithm Algorithm 4
return *nothing*
end function

Algorithm 4 Matrix-Free GPU kernel Action of matrix-free A for west boundary (face 1).

if $2 \le i \le N$ and $j = 1$ **then** ▷ interior west nodes
odata[g] = $\left(M_{rr}^{int} + M_{rs}^{int} + M_{sr}^{int} + M_{s}^{int} + C_1^{int}\right)$(idata)
▷ apply boundary M and C stencils
odata[g+1] = C_1^{int}(idata) ▷ apply interior C stencil
odata[g+2] = C_1^{int}(idata) ▷ apply interior C stencil
end if
if $i = 1$ and $j = 1$ **then** ▷ southwest corner node
odata[g] = $(M_{rr}^{sw} + M_{rs}^{sw} + C_1^{sw})$(idata)
▷ apply southwest partial M and C stencils
odata[g+1] = C_1^{sw}(idata) ▷ apply southwest interior boundary C stencil
odata[g+2] = C_1^{sw}(idata) ▷ apply southwest interior boundary C stencil
end if
if $i = N+1$ and $j = 1$ **then** ▷ northwest corner node
odata[g] = $(M_{rr}^{nw} + M_{rs}^{nw} + C^{nw})$(idata)
▷ apply northwest partial M and C stencils
odata[g+1] = C^{nw}(idata) ▷ apply northwest interior boundary C stencil
odata[g+2] = C^{nw}(idata) ▷ apply northwest interior boundary C stencil
end if

standard sparse matrix format CSR in CUDA.jl, with a focus on integration with preconditioning for improving CG performance.

We set up our benchmark as follows: We discretize the domain $\bar{\Omega}$ in each direction using $N + 1$ grid points, varying N from 2^4 to 2^{13}, so the matrix $\boldsymbol{A}$ is of size $(N + 1)^2 \times (N + 1)^2$. Figure 12 and Figure 13 compare the performance of the matrix-free implementation against the matrix-explicit SpMV provided with cuSPARSE using the CSR format on both the A100 GPU and V100 GPU. The performance is measured by profiling 10,000 SpMV calculations with NVIDIA Nsight Systems, and the time results shown in the figures represent the time to perform one SpMV calculation. For problem sizes large enough for GPUs with N greater than 2^{10}, we see consistent speedup from `mfA!()` kernel with higher speedup achieved for larger problem sizes. On the A100 GPU, our speedup ranges from 3.0× to 3.1×, . On the V100 GPU, we see a similar trend, with our speedup ranging from 3.1× to 3.6×.

The `mfA!`() kernel has a low arithmetic intensity of 0.28 based on the computation of the interior points (which accounts for more than 99% of the total computation and data access). This puts the `mfA!`() kernel in the bandwidth-limited regime Ding and Williams (2019). If we plot this on the Roofline model, as shown in Figure 14 as the left red dot, we see that our kernel achieves performance that is *higher* than what is possible for the given arithmetic intensity. If we calculate the arithmetic intensity based on the assumption that the data is read from the DRAM only *once* (i.e., the ideal case when the kernel only incurs compulsory cache misses), as shown in Figure 14 as the right red dot, we see a higher arithmetic intensity of 1.85 and our achieved performance falls below the Roofline. This suggests that a large portion of our data is coming from the fast memory (e.g., L1 or L2 caches), leading to performance that is better than what can be achieved if the data is only coming from the DRAM.

To confirm our hypothesis, we use NVIDIA Nsight Compute to profile our code for the problem size of N=2^{13}. The profile shows that we achieved 72% L1 cache hit rate and 57% L2 cache hit rate, which indicates that the majority of our data is coming from the L1 and L2 caches (approximately 88%), and that our DRAM reads are due mostly to compulsory cache misses (i.e., when the input data is read for the first time). This explains why our code performs better than the DRAM-bounded performance. Figure 15 shows the Roofline model generated by Nsight Compute, based on performance counter measurements of how much of the overall data is coming

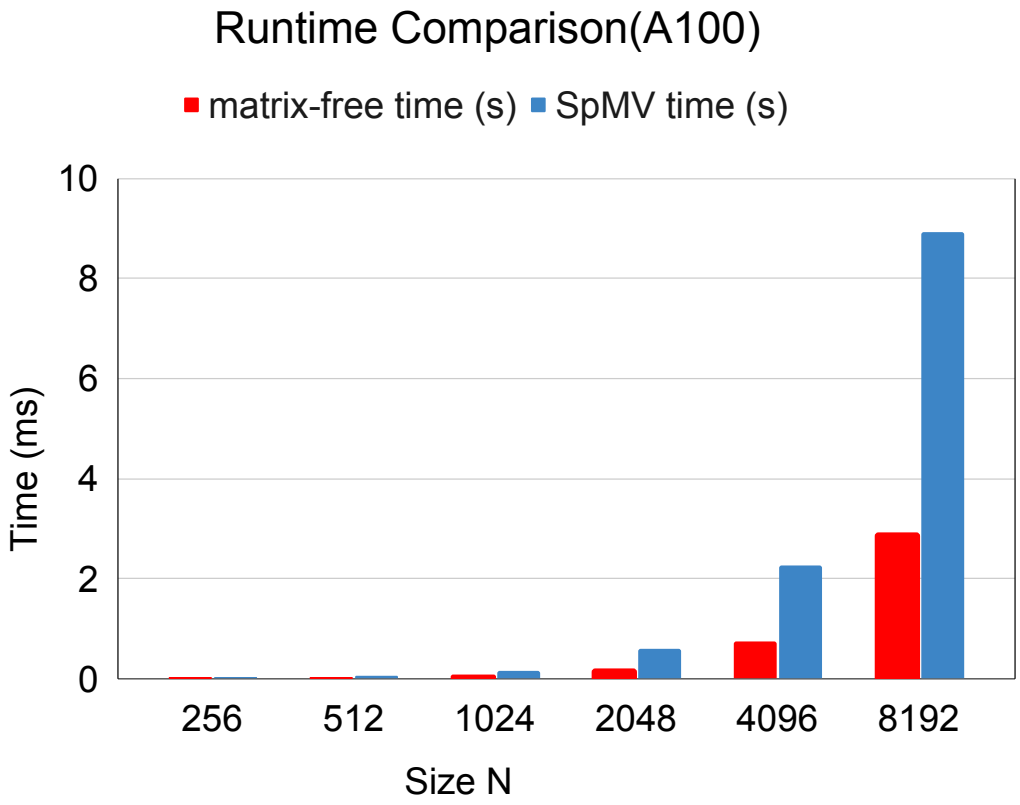

Figure 12. Performance of SpMV vs matrix-free `mfA!`() on A100 GPU. Total time for matrix-free (red) and matrix-explicit CSR (blue) formats are shown in charts plotted against N, where the matrix is size $(N+1)^2 \times (N+1)^2$.

from different levels of the memory hierarchy. Figure 15 confirms that the majority of our data comes from the L1 cache, followed by L2 and DRAM. It also suggests that we can further improve the performance of our `mfA!`() kernel by improving data reuse in the L1 cache, which will yield up to 3.8× speedup.

In future work, we will target improved performance of `mfA!`(), for example through additional memory optimization techniques to improve L1 cache hit rate, especially with respect to its performance on newer architectures. In the present work, however, we focus on utilizing `mfA!`() to solve the linear system with preconditioning.

4.4.2 Memory Usage Comparison. Next we compare the memory usage of `mfA!`() against the SpMV kernel via the built-in memory status function in CUDA.jl. CUDA.jl

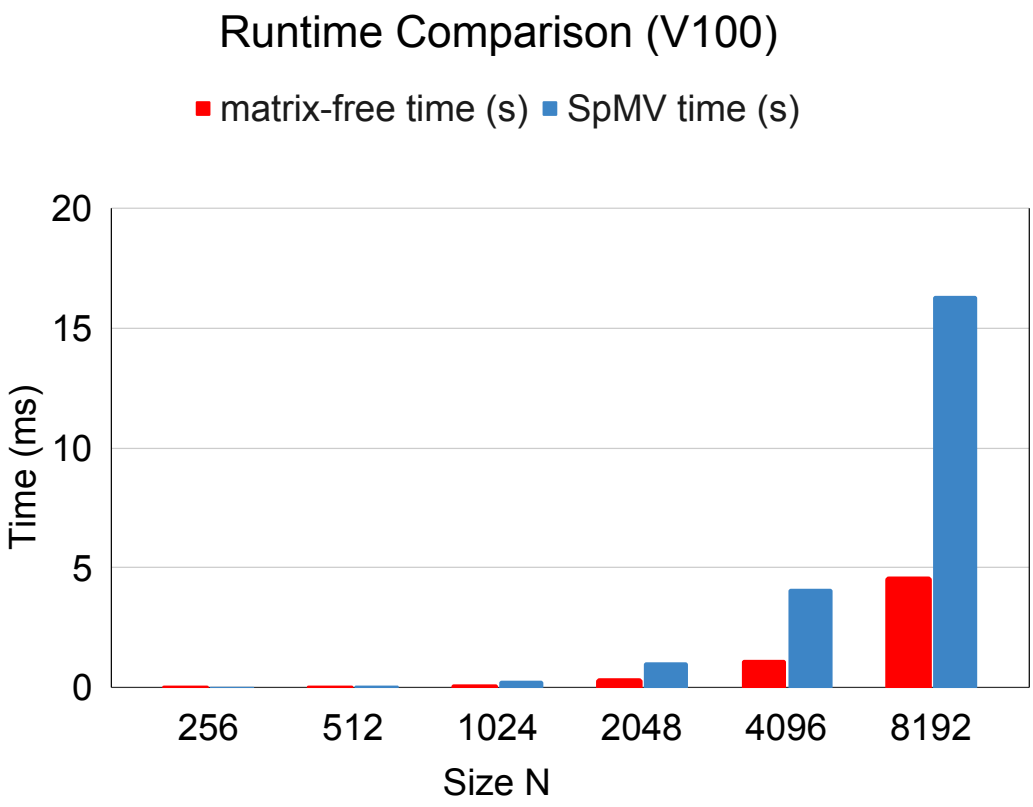

Figure 13. Performance of SpMV vs matrix-free `mfA!`() on V100 GPU. Total time for matrix-free (red) and matrix-explicit CSR (blue) formats are shown in charts plotted against N, where the matrix is size $(N+1)^2 \times (N+1)^2$.

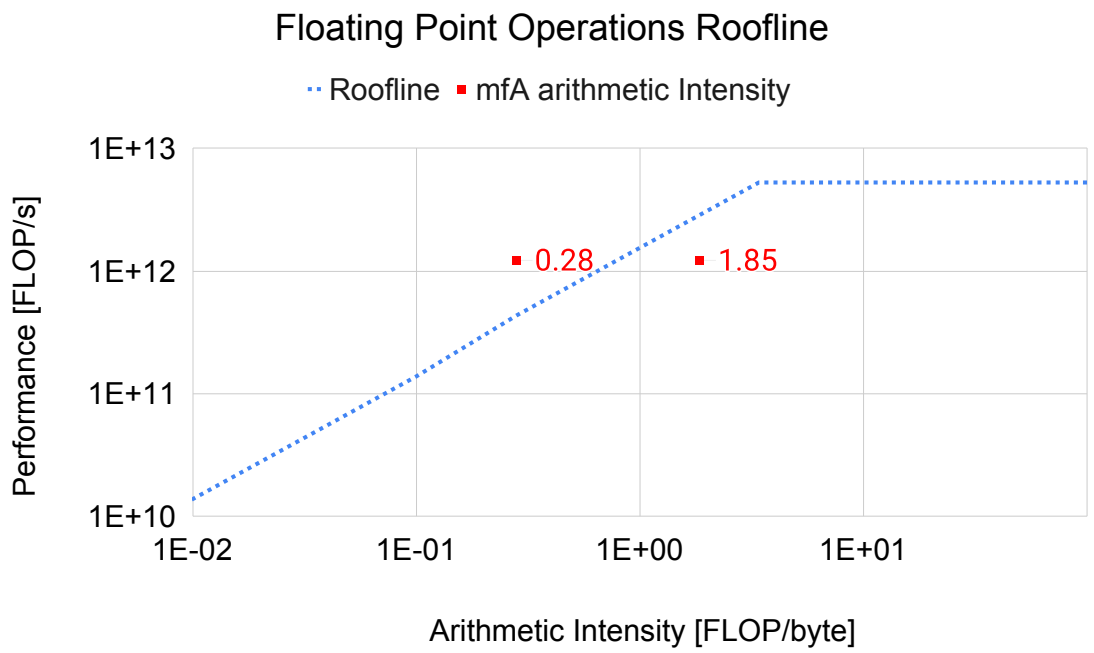

Figure 14. Roofline model analysis for our matrix-free `mfA!()` on the A100 GPU. The red dot on the left represents the performance achieved by our kernel and its arithmetic intensity (0.28). The red dot on the right represents the same but assuming data is loaded only once from DRAM (i.e., compulsory misses), which yields a higher arithmetic intensity (1.85). The fact that our kernel (red dot) achieves higher performance than what is predicted by the Roofline model suggests that a large portion of our data is coming from the caches.

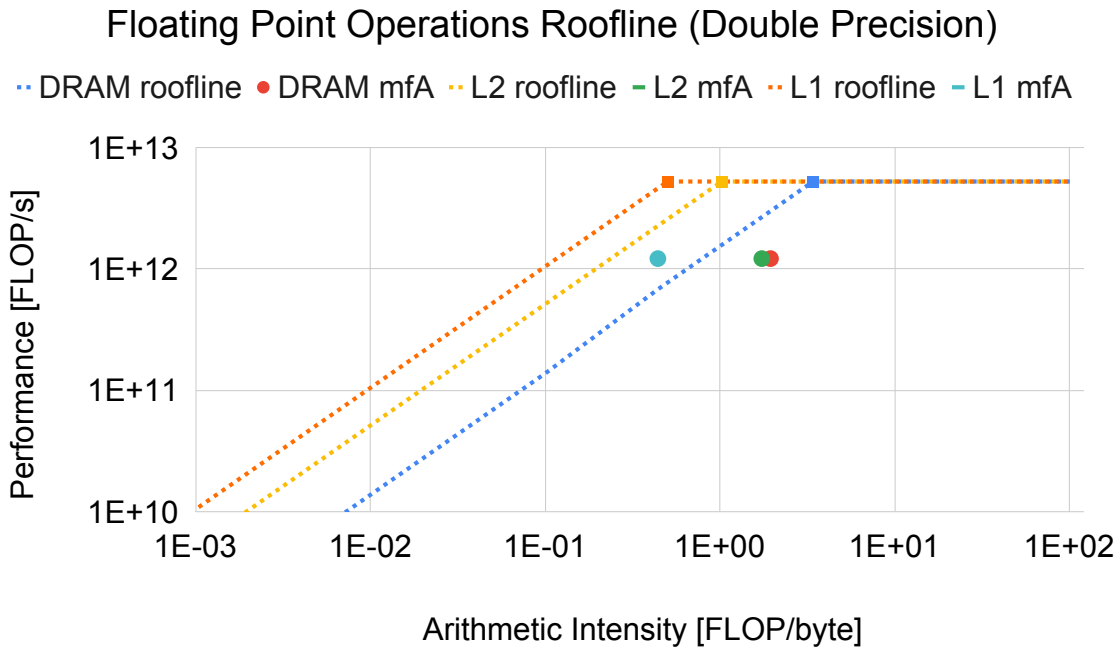

Figure 15. Roofline model generated by Nsight Compute, based on performance counter measurements of how much of the overall data is coming from different levels of the memory hierarchy. This confirms our hypothesis that the majority of our data is coming from the L1 cache, and that further improving data reuse in L1 will yield up to 3.8× speedup.

N	m	nzval	memory size
2^{10}	1050625	9447429	0.1596 GB
2^{11}	4198401	37769221	0.6379 GB
2^{12}	16785409	151035909	2.5509 GB
2^{13}	67125249	604061701	10.2020 GB

Table 2. Number of nonzero values (nzval) for CSC or CSR sparse matrices with different N, where matrix size is $(N+1)^2 \times (N+1)^2$. The matrices are SPD. Here, m represents the number of rows, and nzval represents the number of nonzero values. The total memory size (last column) is calculated using previous columns.

N	crr/css/crs	Ψ_1/Ψ_2	total memory size
2^{10}	0.008405 GB	8 KB	0.02523 GB
2^{11}	0.03359 GB	16 KB	0.1008 GB
2^{12}	0.1343 GB	32 KB	0.4029 GB
2^{13}	0.5370 GB	65 KB	1.6111 GB

Table 3. Memory allocation for matrix-free methods where matrix size is $(N+1)^2 \times (N+1)^2$. Here crr, css, and csr correspond to coefficient matrices of size $(N+1)^2$. Ψ_1 and Ψ_2 are used in Dirichlet boundary conditions and are vectors of length $N+1$. Total memory allocated (last column) is calculated using previous columns.

currently has good support for only three different sparse matrices: CSR, CSC, and COO. In Julia, the default sparse matrix format is CSC, but in CUDA.jl, the default sparse matrix format is CSR, and thus, there is a necessary conversion between these two formats when converting the CPU arrays to GPU arrays in Julia. However, for our problem, where the matrix is SPD, both CSR and CSC formats use exactly the same amount of memory; the only difference is in the use of row pointer `rowptr` values (for CSR) instead of column pointer values `colptr` (for CSC), and the order of nonzero values `nzval`. To avoid redundancy, we merge key results in memory consumption for CSC and CSR formats into three different numbers for each N. The collected data is given in Table 2.

For the matrix-free method, memory consumption is reported in Table 3. In order to perform the matrix-vector product, we need to allocate memory to store the coefficients c_{rr}, c_{ss} and c_{rs}; each requires the same size of memory as the numerical solution and must be stored on each grid level when using geometric multigrid as a preconditioner. In addition, we must compute and store the minimum coefficient values $\boldsymbol{C}_{rr}^{k,min}$ on faces 1 and 2, as specified in (4.7), which we denote $\Psi_1 =$ and Ψ_2, respectively.

Table 4. Iterations and time to converge for $N = 2^{10}$ using 1 smoothing step in PETSc PAMGCG with V cycle (first three rows) vs. our MGCG using Richardson's iteration as smoother (last row)

mg_levels_ksp_type	mg_levels_pc_type	iters	time
chebyshev	sor	18	4.105 s
	jacobi	22	3.382 s
	bjacobi	17	3.945 s
richardson	sor	18	3.581 s
	jacobi	49	3.729 s
	bjacobi	16	3.729 s
cg	sor	17	4.081 s
	jacobi	23	3.849 s
	bjacobi	16	3.971 s
richardson	none	11	0.086 s

These are associated with Dirichlet boundary conditions and are significantly smaller in size, and thus reported in KB. Adding up these contributions, we can compute the total memory size, which we provide in the last columns of Tables Table 2 and Table 3: We can see that there is a significant reduction in additional required memory for the matrix-free method than the memory to store sparse matrices in CSC or CSR format. When calculating the total memory used for an SpMV operation (including writing results into output vectors), we need to add additional memory allocated for the input data and output data, which require the same memory as the coefficients (the first column of Table 3). A simple calculation can show that the total memory required when using an SpMV kernel is a constant 4.2× of that required for the matrix-free method.

4.5 Performance: Matrix-free MGCG

4.5.1 Preconditioning performance. MG methods have many tunable parameters. For this initial study, we explored the MGCG performance varying the number of Richardson pre- and post-smoothing steps on every level between 1 and 5. We considered a single V-cycle (i.e. taking $\nu_1 = \nu_2 = \nu_3 = 5$ and $N_{maxiter} = 1$ in Algorithm 1), including on the coarsest grid (5 grid points in each direction). For all tests in this work we initialize the iterative scheme with the zero vector. All algorithms stop when the relative residual is reduced to less than 10^{-6} times the initial residual.

Comparisons against an unpreconditioned CG are not generally appropriate as most real-world applications require preconditioning to make any solution tractable. The Portable,

Table 5. Iterations and time to converge for $N = 2^{10}$ using 5 smoothing steps in PETSc PAMGCG with V cycle (first three rows) vs. our MGCG using Richardson's iteration as smoother (last row)

mg_levels_ksp_type	mg_levels_pc_type	iters	time
chebyshev	sor	10	10.76 s
	jacobi	14	10.20 s
	bjacobi	9	10.58 s
richardson	sor	9	10.13 s
	jacobi	DV	9.24 s
	bjacobi	8	10.28 s
cg	sor	9	10.47 s
	jacobi	13	10.54 s
	bjacobi	8	10.45 s
richardson	none	8	0.069s

Extensible Toolkit for Scientific Computing (PETSc) Balay et al. (2023) is one of the most widely used parallel numerical software libraries, featuring extensive preconditioning methods, many of which can be tested by users via relatively simple command-line options. We experimented with several of PETSc's off-the-shelf algebraic multigrid preconditioned CG solvers (denoted PAMGCG). PETSc's PAMGCG is similar to our MGCG and only requires loading PETSc formatted **A** and **b** (from which it forms the coarse grid operators).

We tested PAMGCG with various configurations against our custom MGCG, applying the same stopping criterion (here based on the relative norm of the residual vector reduced to 1E-6), with results provided in Table 4 and Table 5. The `mg_levels_ksp_type` and `mg_levels_pc_type` in the tables stand for Krylov subspace method types and preconditioner types used at each level of the multigrid in PAMGCG. When classical iterative methods are used as smoothers, `mg_levels_ksp_type` is set as `richardson` and the particular smoother (e.g. Jacobi) is set by `mg_levels_pc_type`. Since our MGCG uses Richardson's iteration as the smoother for multigrid, we report `mg_levels_ksp_type` as `richardson` and `mg_levels_pc_type` as `none` to maintain coherence across the columns. Iterations and total time to converge are reported. We found that the Jacobi iteration is not a good choice as smoother in PAMGCG. When using 1 smoothing step, it takes more iterations than other configurations. It does not converge (denoted as DV) when using 5 smoothing steps. Aside from this configuration, other PETSc configurations in the table exhibit comparable performance in both the number of

Table 6. Time to perform a direct solve after LU factorization on CPUs vs PCG on GPUs. We report time in seconds and iterations to converge. For AmgX, we report setup + solve time. For our MGCG, setup time is negligible. "ns" is short for the number of smoothing steps. GPU results are tested on A100.

N	Direct Solve	AmgX (ns = 1)	AmgX (ns = 5)
2^{10}	0.912 s	(0.0319 s + 0.0243 s) / 25	(0.0321 s + 0.0435 s) / 17
2^{11}	6.007 s	(0.086 s + 0.161 s) / 55	(0.086 s + 0.311 s) / 38
2^{12}	22.382 s	(0.310 s + 0.235 s) / 24	(0.323 s + 0.488 s) / 15
2^{13}	134.697 s	(1.334 s + 1.643 s) / 24	(1.217 s + 1.865 s) / 16

Table 7. Time to perform a direct solve after LU factorization on CPUs vs PCG on GPUs. We report time in seconds and iterations to converge. For AmgX, we report setup + solve time. For our MGCG, setup time is negligible. "ns" is short for the number of smoothing steps. GPU results are tested on A100.

N	Direct Solve	SpMV-MGCG (ns = 5)	MF-MGCG (ns = 5)
2^{10}	0.912 s	7.019E-2 s / 8	2.851E-2 s / 8
2^{11}	6.007 s	0.158 s / 7	0.0605 s / 7
2^{12}	22.382 s	0.564 s / 7	0.207 s / 7
2^{13}	134.697 s	5.028 s / 7	0.865 s / 7

iterations and the convergence time. We found that additional options within PAMGCG play relatively minor roles in performance. Our MGCG results (reported in the last rows), however, show superior performance in terms of both iteration counts and overall time.

4.5.2 Performance on GPUs. With the matrix-free action of $\boldsymbol{A}$ established, we can solve system Equation 4.9 with a matrix-free version of our custom MGCG method (MF-MGCG). Other than low-level GPU kernels, Julia also supports high-level vectorization for GPU computing, which we utilize extensively in our MGCG code for convenience. In this section, we compare its performance against MGCG using the cuSPARSE (matrix-explicit) SpMV (SpMV-MGCG) and also against the state-of-the-art off-the-shelf methods offered by NVIDIA, namely, AmgX - the GPU accelerated algebraic multigrid. The solvers and preconditioners used by AmgX are stored as JSON files. We explored different sample JSON configuration files for AmgX in the source code and found that CG preconditioned by classical AMG performed best for our problem. To maintain a multigrid setup comparable to our MGCG, we modified the `PCG_CLASSICAL_V_JACOBI.json` to use 1 and 5 smoothing steps with block Jacobi as the smoother. All algorithms stop when the relative residual is reduced to less than 10^{-6} times the initial residual. We report results from AmgX in Table 6 and our MGCG in Table 7. Also included in the table are results using a direct solve (using LU factorization in LAPACK in

Julia) only because it is so often used in the earthquake cycle community for volume based codes B. A. Erickson et al. (2020a) and our developed methods offer promising alternatives. As illustrated, the GPU-accelerated iteratives schemes achieve much better performance for the problem sizes tested.

The results from Table 6 and Table 7 show that our MGCG method uses fewer iterations to converge compared to AmgX, while iterations for both remain generally constant with increasing problem size. When we increase smoothing steps from 1 to 5, the AmgX sees reduced iterations, but the time to solve also increases by roughly $3\times$. Because we apply rediscretization (rather than Galerkin coarsening) for MGCG, the setup time is negligible. The setup time in the AmgX is comparable to the solve time however, which adds additional cost to use AmgX as a solver. Our SpMV-MGCG is roughly $2\times$ slower than the AmgX using 1 smoothing step, but our MF-MGCG is faster than AmgX, up to $2\times$ speedup for $N = 2^{13}$. Compared to our SpMV-MGCG, our MF-MGCG achieves more than $2\times$ speedup, and the speedup is more obvious at $N = 2^{13}$, indicating that the MF-MGCG is suitable for large problems.

In this chapter, we present a matrix-free implementation of the multigrid preconditioned conjugate gradient in order to solve 2D, variable coefficient elliptic problems discretized with an SBP-SAT method. Our customed multigrid preconditioner achieves similar preconditioning performance against the multigrid using Galerkin's condition from previous work, and it is more suitable for GPU code. The MGCG algorithm requires a nearly constant number of iterations to converge for various problem sizes. We used Nsight Compute to analyze the performance of our matrix-free kernel. This offers us more insights into the achieved computation and memory performance, which points to directions for future kernel-level optimizations on newer GPU architectures.

This work is a fundamental first step towards high-performance implementations to solve linear systems using SBP-SAT methods. Future work will target SBP-SAT methods with higher-order accuracy in 3D, as well as explorations of additional GPU kernel optimization and multi-GPU implementation. We also plan to improve the performance of the preconditioner by systematic experiments with different preconditioner configurations using PETSc and applying second-order smoothers that have exhibited improved performance in the multigrid method as well

as the mixed-precision techniques Abdelfattah et al. (2021); Golub and Varga (1961); Gutknecht and Röllin (2002).

CHAPTER V

SEAS BENCHAMRK PROBLEMS

5.1 SEAS problems

5.1.1 Modeling Environment. The applications are motivated by the study of quasi-static deformation of the solid Earth over the time scales of earthquake cycles. In both the interseismic and coseismic phases, the off-fault material response is modeled as elastic-plastic.

$$\rho\ddot{u} = \nabla \cdot \sigma + F, \sigma = \boldsymbol{C} : (\epsilon - \epsilon^p) \tag{5.1}$$

Here, ρ is the material density, u is the vector of particle displacements, F is the body fordce, $\boldsymbol{C}$ is the stiffness tensor of elastic moduli, and ϵ and ϵ^p are the elastic and plastic strains. A fault network (shown in Figure 16) is composed of faults governed by non-linear, rate-and-state friction which determines the relationship between the slip velocity V to shear traction τ with the (effective) norm stress $\bar{\sigma}$, the friction coefficient f and a state variable Ψ.

$$\tau = \bar{\sigma} f(V, \Psi), \Psi = G(V, \Psi) \tag{5.2}$$

The form of the state evolution law G can take several forms such as the aging law in which the state evolves in the absence of slip or the slip law with strong rate-weakening.

During the interseismic phase, the inertial terms in the governing equation 5.1 are set 0 ($\ddot{u} = 0$). Tectonic loading is imposed through time-dependent boundary conditions and the slip on faults are incorporated through friction law 5.2. The evolution of Ψ constraints the time step, and very large time steps can be used during the interseismic phase. The main computational challenge comes from solving the large linear systems of equations that come from the discretization of the steady-state version of the equation 5.1. During the interseismic phase, tectonic loading determines the boundary conditions and the stress on faults as the result of elastic deformation. Once an event begins to nucleate, we enter the coseismic phase and inertial terms of governing equation 5.1 are retained. It is more efficient to use explicit integration during this period because it simplifies the computation. In both phases, the governing equation can be solved using the SBP-SAT methods mentioned in the previous section.

5.1.2 3D Problem Setup. The 3D problem setup described below is taken from previous SEAS publications B. A. Erickson et al. (2020b). The medium is assumed to be a

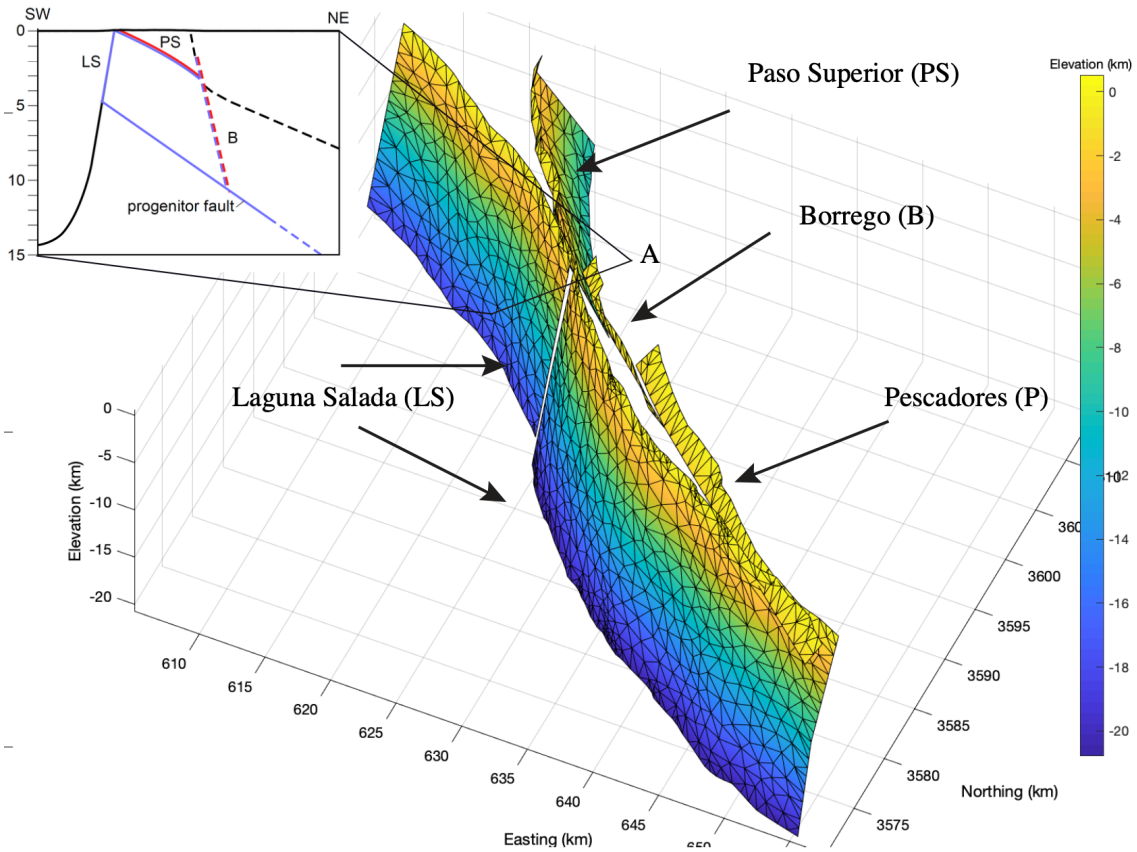

Figure 16. A 3D image of the complex fault network from EMC earthquake; image generated using scripts from Marshall et al. (2017)

homogeneous, isotropic, linear elastic half-space defined by

$$\mathbf{x} = (x_1, x_2, x_3) \in (-\infty, \infty) \times (\infty, \infty) \times (0, \infty) \tag{5.3}$$

with a free surface at $x_3 = 0$ and x_3 as positive downward. A vertical, strike-slip fault is embedded at $x_1 = 0$, We use the notation "+" and "-" to refer to the different sides of the fault.

We assume 3D motion, letting $u_i = u_i(\mathbf{x}, t), i = 1, 2, 3$ denote the displacement in the i-direction.

Hooke's law for linear elasticity is given by

$$\sigma_{ij} = K\epsilon_{kk}\delta_{ij} + 2\mu(\epsilon_{ij} - \frac{1}{3}\epsilon_{kk}\delta_{ij}) \tag{5.4}$$

for bulk modulus K and shear modulus μ. The strain-displacement relations are given by

$$\epsilon_{ij} = \frac{1}{2}\left[\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right] \tag{5.5}$$

The description of these benchmark problems can be found in B. Erickson and Jiang (2018); Jiang and Erickson (2020).

To simulate the SEAS problems using the quasi-static method, it usually follows these steps.

Algorithm 5 Quasi-static Formulation Algorithm

1: **Step 1:** Initialize boundary conditions and state variables
2: **while** simulation time not reached **do**
3: **Step 2:** Solve steady-state problems to obtain displacements
4: In 2D, Solve Poisson's equation
5: In 3D Solve linear elasticity
6: **Step 3:** Calculate stress from displacements
7: **Step 4:** Calculate displacement velocity using stress and rate-and-state frictions
8: **Step 5:** Determine time step size (dt) using ODE solver
9: **Step 6:** Calculate state variables using aging law and dt
10: **Step 7:** Update boundary conditions using velocity and dt
11: **end while**

The DifferentialEquations.jl package provides powerful adapted ODE solvers based on Runge-Kutta methods and useful ODE interfaces that allow us to modify data and write to outputs. The key challenge here is step 2 which requires solving a large linear system that is formed with the SBP-SAT methods. It's difficult to apply direct methods due to their high memory requirements and computational inefficiency. In the next two chapters, we will go into detail to first formulate the linear systems using the SBP-SAT methods and then apply HPC algorithms to solve such problems.

5.1.3 Solving for rate-and-state friction. Rate-and-state friction plays a central role in all SEAS problems. The friction coefficient function f in SEAS problems is given as a regularized formulation

$$f(V,\theta) = a \sinh^{-1}[\frac{V}{V_0} \exp \frac{f_0 + b \ln(V_0\theta/L)}{a}] \tag{5.6}$$

f_0 represents the reference friction coefficient. V_0 represents slip rate, and a and b are rate-and-state parameters. For benchmark problem 1 and 5, b is constant as b_0, but a varies throughout computational domain Ω_f in order to define velocity-weakening and velocity-strengthening regions. We will define them in respective sections differently.

The state variable θ evolves according to the aging law

$$\frac{d\theta}{dt} = 1 - \frac{V\theta}{L} \tag{5.7}$$

The fault strength is given as

$$\mathbf{F} = \bar{\sigma}_n f(V,\theta)\frac{\mathbf{V}}{V} \tag{5.8}$$

where $\mathbf{F}$ and $\mathbf{V}$ are vectors and V is the norm of the $\mathbf{V}$. The rate-and-state friction where shear stress on fault is equal to fault strength $\mathbf{F}$. In Quadsi-static simulations, fault displacements are solved given governing equations and boundary conditions. Shear stress is calculated using displacements on fault. In equation 5.8 and 5.6, we can solve for V and then calculate components of V based on components of $\mathbf{F}$. This significantly simplifies the calculation and improves numerical accuracy due to the magnitude differences between different components of $\mathbf{F}$ and $\mathbf{V}$.

Once all parameters in Equation 5.6 and 5.8 are known along shear stress calculated from displacements, it is common to apply Newton's method given in Algorithm 6 to solve the non-linear equation to obtain V.

In our problem with high nonlinearity from the $\sinh^{-1}$ function, to improve numerical stability, we also need to apply the "safe-guarded" method. One commonly used method is called bisection, it uses a similar approach to binary search. Based on the functional values of a close range $[x_L, x_r]$, it updates the search range of the root x. Newton's method modified with Bisection is given in Algorithm 7.

Because V is solved for each grid point independently, the above methods can be implemented using vectorized approach either on CPUs or GPUs. Logical operations used in the control branch can be implemented using *masks*, which is a common technique in parallelization.

Algorithm 6 Newton's Method

Initialize x_0
Set tolerance ϵ
Set maximum number of iterations $N_{\max}$
for $k = 0, 1, 2, \ldots, N_{\max}$ **do**
    Compute $f(x_k)$ gradient $\nabla f(x_k)$
    Determine search direction $d_k = -(\nabla f(x_k))^{-1} f(x_k)$
    Perform line search to find step size α_k such that $f(x_k + \alpha_k d_k) < f(x_k)$
    Update $x_{k+1} = x_k + \alpha_k d_k$
    if $\|\nabla f(x_{k+1})\| < \epsilon$ **then**
        Convergence achieved
        break
    end if
end forreturn x_{k+1}

Algorithm 7 Newton's Method with Bisection

Initialize x_0, bounds $[x_L, x_R]$, and $x = (x_L + x_R)/2$
Set tolerance ϵ_a, ϵ_r and step size $\alpha_k = x_R - x_L$
Set maximum number of iterations $N_{\max}$
for $k = 0, 1, 2, \ldots, N_{\max}$ **do**
    Compute $f(x_k)$ and gradient $\nabla f(x_k)$
    Compute f_L, f_R as $f(x_L), f(x_R)$
    Determine search direction $d_k = -(\nabla f(x_k))^{-1} f(x_k)$
    Perform line search to find step size α_k such that $f(x_k + \alpha_k d_k) < f(x_k)$
    Update $x_{k+1} = x_k + \alpha_k d_k$
    if $x_k < x_L$ or $x_k > x_R$ **then**
        $x_k = (x_L + x_R)/2$
        $\alpha_k = (x_R - x_L)/2$
    end if
    if $f(x_k) * f_L > 0$ **then**
        $(f_L, x_L) = (f, x_k)$
    else
        $(f_R, x_R) = (f, x_k)$
    end if
    if $\|\nabla f(x_{k+1})\| < \epsilon_a$ and $\|\alpha_k\| < \epsilon_a + \epsilon_r * (\|\alpha_k\| + \|x\|)$ **then**
        Convergence achieved
        break
    end if
end forreturn x_{k+1}

The above calculations, although complex in numerical form and involving key concepts in earthquake cycle simulations and rate-and-state friction laws are actually very cheap and can be accelerated easily using vectorized operations. It is not the focus of this thesis.

5.1.4 Methods of Lines. Governing equations in SEAS problems, like many other PDEs, involve both time and space. Methods of Lines (MOL) is a common approach to solving these PDEs. The basic idea of MOL is to replace the spatial derivatives in PDE with algebraic approximations such as the SBP-SAT method in our work. Once this is done, the spatial derivatives are no longer explicitly dependent on spatial independent variables. Thus, the only variable left is t. In other words, we have a system of ODEs that approximate the original PDE and use various ODE solvers to solve the original PDE. Since ODE solver is not the focus of this thesis, we use the default and the mostly recommended ODE solver `tsit5()` provided in DifferentialEquations.jl Tsitouras (2011). This is an adaptive ODE solver based on the Runge-Kutta pair of orders 5(4). The numerical stability of ODE solvers plays an important role in numerical solutions to PDEs, and they affect the simulation results and running time significantly in our research. However, this thesis is on the spatial discretization part of the MOL using the SBP-SAT method, which we will discuss in the next section in detail for BP1 and BP5 separately. Contents related to ODE solvers will only be briefly mentioned without detailed discussion and analysis.

5.2 BP1-QD problem

5.2.1 Problem description. The problem setup is similar to the 3D problem setup described in section 5.1. Here, we assume antiplane shear motion that is invariant in the y-direction. The displacement vector $\mathbf{u} = \mathbf{u}(x, y, z)$, and the only non-zero component of the displacement vector is in the y-direction. We use the scalar value u to denote this displacement component.

The 3D problem is then reduced to a 2D problem

$$0 = \frac{\partial \sigma_{xy}}{\partial x} + \frac{\partial \sigma_{yz}}{\partial z} \tag{5.9}$$

in the domain $(x, z) \in (-\infty, \infty) \times (0, \infty)$, where

$$\sigma_{x,y} = \mu \frac{\partial u}{\partial x}; \quad \sigma_{yz} = \mu \frac{\partial u}{z} \tag{5.10}$$

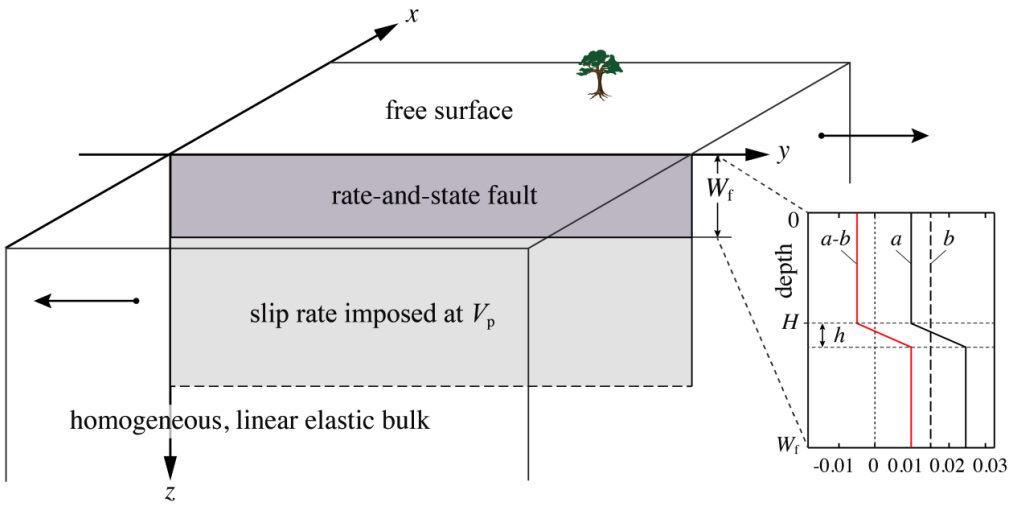

Figure 17. BP1 considers a planar fault embedded in a homogeneous, linear elastic halfspace with a free surface. The fault is governed by rate-and-state friction down to the depth W_f and creeps at an imposed constant rate V_p down to the infinite depth. The simulations will include the nucleation, propagation, and arrest of earthquakes, and aseismic slip in the post- and inter-seismic periods. The figure and the description are given in B. Erickson and Jiang (2018)

The above is essentially a Poisson's equation that we solve in Section 4. The formulation of the linear system for the BP1-QD problem is similar to what is described in Chapter 4. Instead of using methods of manufactured solutions to verify our solution and convergence, we verify our results via comparison with results from the simulations of other researchers. The formulation of the linear systme for BP1-QD problem is similar to what is described in Chapter . Instead of using methods of manufactured solutions to verify our solution and convergence, we verify our results via comparison with results from the simulations of other researchers.

Most parameters and boundary and initial conditions are given in the SEAS problem description. In this problem, a varies with the depth

$$a(z) = \begin{cases} a_0, & 0 \le z < H \\ a_0 + (a_{\max} - a_0)(z - H)/h, & H \le z < H + h \\ a_{\max}, & H + h \le z < W_f \end{cases} \tag{5.11}$$

Below depth W_f, the fault creeps at an imposed constant rate, given by the interface condition

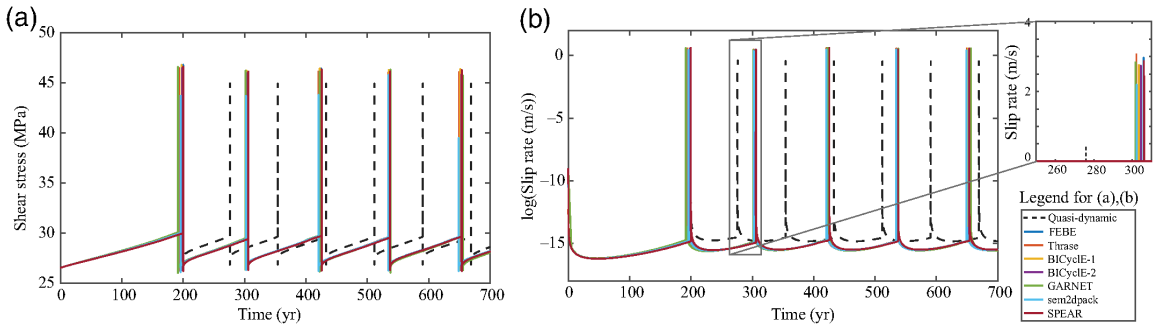

Figure 18. Long-term behavior of BP1-FD models. Our model name is Thrase in this figure. (a) Shear stress and (b) slip rates at the depth of 7.5 km across codes with sufficiently large computational domain sizes. Also shown (in dashed black) are those for the quasi-dynamic counterpart BP1-QD. The color version of this figure is available only in the electronic edition. B. A. Erickson et al. (2023)

$$V(z,t) = V_p, \quad z \geq W_{\mathrm{f}} \tag{5.12}$$

For the simulation, we discretize our problem on a 401 $\times$ 401 grid points in each direction after discretization on a 2D domain with around 160k unknowns. The conjugate gradient method without a preconditioner on GPUs is fast enough for the simulation to be complete in $\sim$ 2 days on a V100 GPU after solving linear system $\sim$ 300000 times. It takes around 0.5 seconds to solve linear systems, update values, and perform other data operations. The results for this simulation are published in B. A. Erickson et al. (2023) under the model name Thrase.

The Figure 18

Figure 18 shows our results along with simulation results from other researchers. When using different methods to solve the same benchmark problem, we achieve comparable results regarding the time between two slip events and the slip rate.

For the 2D problem with more than 1000 grid points in each direction or a 3D problem with recommended resolutions from benchmark problem 5, we will be solving linear systems with millions or tens of millions of unknowns. The above approach is too slow even on the latest GPUs. We need more advanced algorithms like the MGCG method described in Chapter 5.

5.3 BP5-QD problem

5.3.1 Problem description.

5.3.2 Boundary and Interface conditions.

At $x_1 = 0$, the fault defines the interfaces. A free surface lies at $x_3 = 0$, where all components of traction have 0 value. This

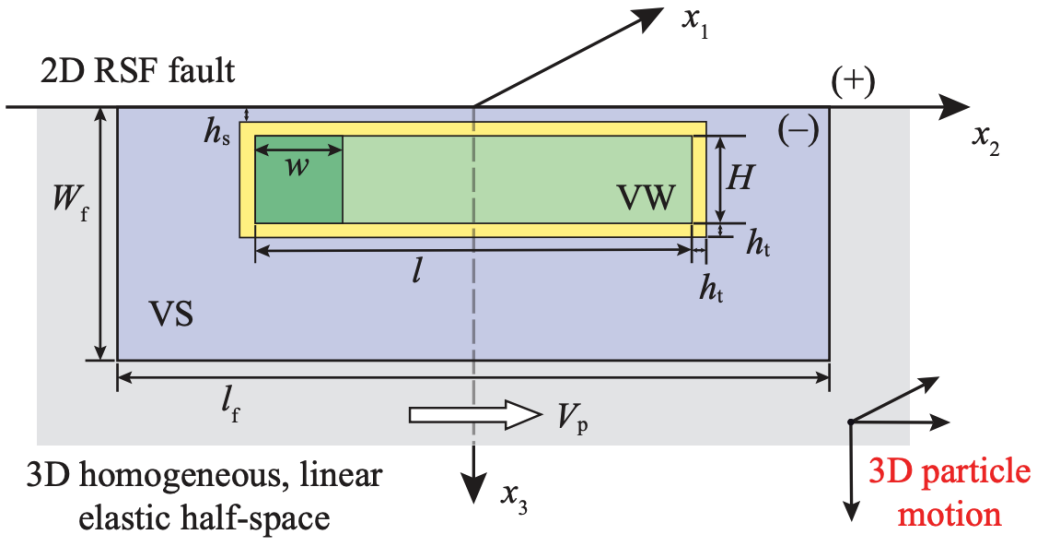

Figure 19. This benchmark considers 3D motion with a planar fault embedded vertically in a homogeneous, linear elastic whole-space. The fault is governed by rate-and-state friction in the region $0 \leq x_3 \leq W_f$ and $|x_2| \leq l_f/2$, outside of which it creeps at an imposed constant horizontal rate Vp (gray). The velocity weakening region (the rectangle in light and dark green; $h_s + h_t \leq x_3 \leq h_s + h_t + H$ and $|x_2| \leq l/2$) is surrounded by a transition zone (yellow) of width ht to velocity strengthening regions (blue). A favorable nucleation zone (dark green square with width w) is located at one end of the velocity-weakening patch. Jiang and Erickson (2020)

condition is written in the following form

$$\sigma_{j3}(x_1, x_2, 0, t) = 0, \quad j = 1, 2, 3 \tag{5.13}$$

We assume a "non-opening condition" on the fault

$$u_1(0^+, x_2, x_3, t) = u_1(0^-, x_2, x_3, t) \tag{5.14}$$

For $j = 2, 3$, we define the slip vector as the jump in horizontal and vertical displacements across the fault.

$$s_j(x_2, x_3, t) = u_j(0^+, x_2, x_3, t) - u_j(0^-, x_2, x_3, t), \quad j = 2, 3 \tag{5.15}$$

We require that components of the traction vector be equal and opposite across the fault, which yields three conditions

$$-\sigma_{11}\left(0^+, x_2, x_3, t\right) = -\sigma_{11}\left(0^-, x_2, x_3, t\right), \tag{5.16}$$

$$\sigma_{21}\left(0^+, x_2, x_3, t\right) = \sigma_{21}\left(0^-, x_2, x_3, t\right), \tag{5.17}$$

$$\sigma_{31}\left(0^+, x_2, x_3, t\right) = \sigma_{31}\left(0^-, x_2, x_3, t\right), \tag{5.18}$$

We denote these three common values as σ (positive means compression), τ and τ_z respectively. In the simulation, τ and τ_z are key values calculated from the displacements and are used in the rate-and-state friction. We also export the log of these two values to the output files.

Most parameters are given in the problem description. The key value is the rate-and-state friction a, given in the following form

$$a(x_2, x_3) = \begin{cases} a_0, & (h_s + h_t \le x_3 \le h_s + h_t + H) \cap (|x_2| \le l/2) \\ a_{\max}, & (0 \le x_3 \le h_s) \cup (h_s + 2h_t + H \le x_3 \le W_f) \\ & \cup (l/2 + h_t \le |x_2| \le l_f/2) \\ a_0 + r(a_{\max} - a_0), & \text{other regions} \end{cases} \tag{5.19}$$

where $r = \max(|x_3 - h_s - h_t - H/2| - H/2, |x_2| - l/2)/h_t$.

Outside the domain Σ_f ($|x_3| > W_\mathrm{f}$ or $|x_2| > l_\mathrm{f}/2$ as denoted in the grey region in the Figure 19), the fault creeps horizontally at an imposed constant rate

$$V_2(x_2, x_3, t) = V_\mathrm{p} \tag{5.20}$$

$$V_3(x_2, x_3, t) = 0 \tag{5.21}$$

where V_{p} is the plate rate.

We also need to specify the initial conditions for the simulation. We assume that slip on the fault separating identical materials does not change normal traction, so σ_n remains constant.

The initial state and prestress on the fault is chosen so that the model can start with a uniform fault slip rate, given by $\mathbf{V} = [V_{\mathrm{init}}, V_{\mathrm{zero}}]$ where V_{zero} is chosen as $10^{-20} m/s$ to avoid infinite log values in the output, and $\boldsymbol{\tau}^0 = \tau^0 \mathbf{V}/V$.

The initial state variable is chosen as the steady state at slip rate V_{init} over the entire fault

$$\theta(x_2, x_3, 0) = L/V_{\mathrm{init}} \tag{5.22}$$

For the BP5-QD problem, we also need to specify an initial value for the slip, which we set to be zero.

$$s_j(x_2, x_3, t) = 0, j = 2, 3 \tag{5.23}$$

The scalar pre-stress τ^0 is chosen as the steady-state stress

$$\tau^0 = \bar{\sigma}_{\mathrm{n}} a \sinh^{-1}[\frac{V_{\mathrm{init}}}{2V_0} \exp(\frac{f_0 + b \ln(V_0/V_{\mathrm{init}}}{a})] + \eta V_{\mathrm{init}} \tag{5.24}$$

To break the symmetry of the problem and facilitate comparisons of different simulations, we choose a small region as a favorable location for nucleation to impose a smaller critical slip distance ($L = 0.13m$) and higher initial slip rate along the x_2-direction ($V_i = 0.03 m/s$) while keeping the initial state variable $\theta(x_2, x_3, 0)$ unchanged. The means we impose higher pre-stress along the x_2-direction.

We use the recommended parameters from the problem description and perform initial simulations for 1800 years.

5.3.3 SBP-SAT formulations for BP5-QD. For this 3D problem, we use SBP-SAT methods similar to Almquist and Dunham (2021) to formulate our linear system. To solve the linear elasticity in 3D, we need to solve for the displacements in x, y, and z directions for each point. We denote the displacement vector as $\boldsymbol{u} = [u_1, u_2, u_3]$. To turn the tensor formulation from Almquist and Dunham (2021) into a matrix formulation for iterative solvers, we first stack the displacements for a point in x, y, z directions, and then for all points in x, y, and z directions. We label the faces for 3D simulation in the following order as shown in Figure 20

The first step is to derive values for σ tensor in 3D.

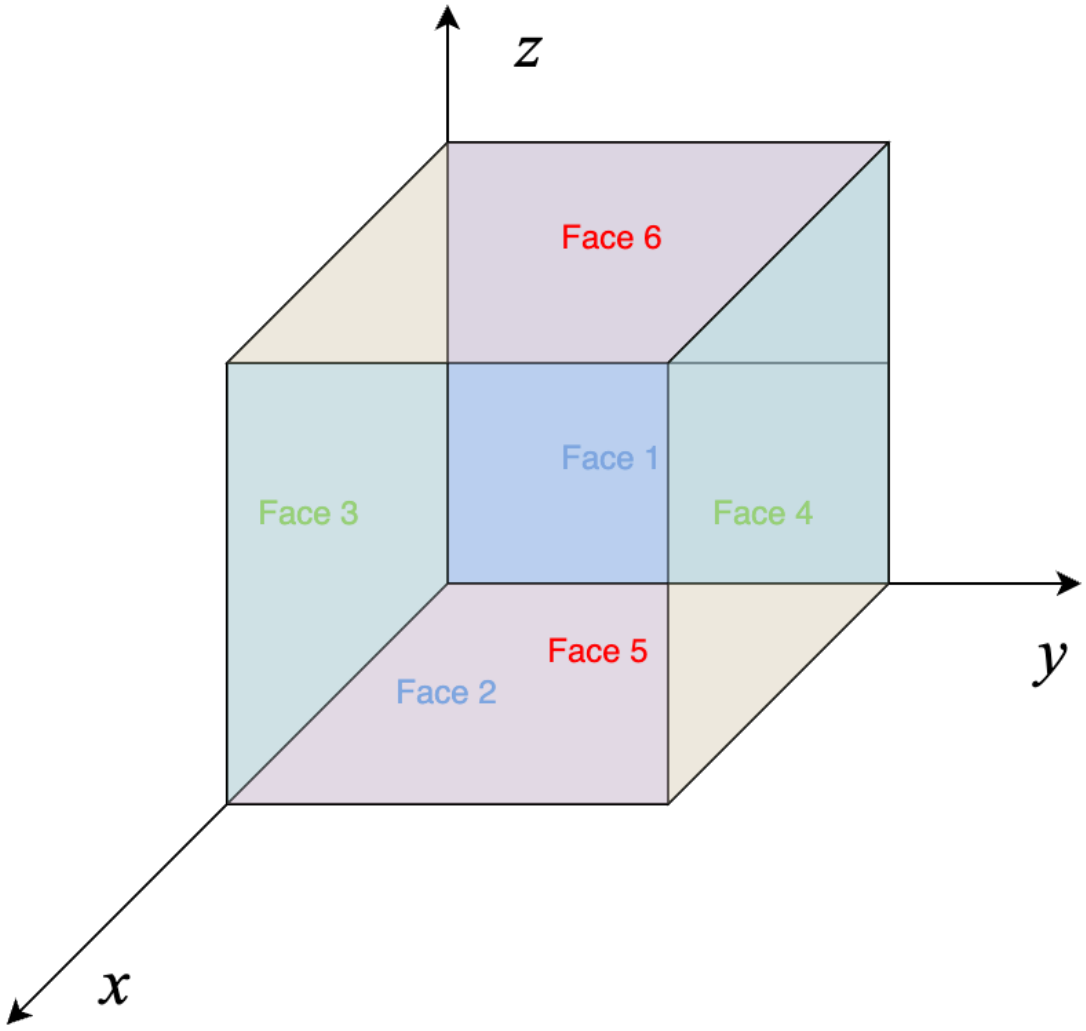

Figure 20. We set up the 3D coordinate and denote faces 1 to 6 using different colors. Face 1 and Face 2 are perpendicular to the x-axis denoted using blue color. Face 3 and Face 4 are perpendicular to the y-axis denoted using green color. Face 5 and Face 6 are perpendicular to the z-axis denoted using red color. We impose Dirichlet boundary conditions on Face 1 and Face 2. For Face 3 to Face 6, we impose traction-free (Neumann) condition.

$$\sigma_{11} = K\epsilon_{kk} + 2\mu(\epsilon_{11} - \frac{1}{3}\epsilon_{kk}) = (K - \frac{2}{3}\mu)(\frac{\partial u_1}{\partial x_1} + \frac{\partial u_2}{\partial x_2} + \frac{\partial u_3}{\partial x_3}) + 2\mu\frac{\partial u_1}{\partial x_1} \tag{5.25}$$

$$\sigma_{12} = 2\mu\epsilon_{12} = \mu(\frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1}) \tag{5.26}$$

$$\sigma_{13} = 2\mu\epsilon_{13} = \mu(\frac{\partial u_1}{\partial x_3} + \frac{\partial u_3}{\partial x_1}) \tag{5.27}$$

$$\sigma_{21} = 2\mu\epsilon_{21} = \mu(\frac{\partial u_2}{\partial x_1} + \frac{\partial u_1}{\partial x_2}) \tag{5.28}$$

$$\sigma_{22} = K\epsilon_{kk} + 2\mu(\epsilon_{22} - \frac{1}{3}\epsilon_{kk}) = (K - \frac{2}{3}\mu)(\frac{\partial u_1}{\partial x_1} + \frac{\partial u_2}{\partial x_2} + \frac{\partial u_3}{\partial x_3}) + 2\mu\frac{\partial u_2}{\partial x_2} \tag{5.29}$$

$$\sigma_{23} = 2\mu\epsilon_{13} = \mu(\frac{\partial u_2}{\partial x_3} + \frac{\partial u_3}{\partial x_2}) \tag{5.30}$$

$$\sigma_{31} = 2\mu\epsilon_{31} = \mu(\frac{\partial u_3}{\partial x_1} + \frac{\partial u_1}{\partial x_3}) \tag{5.31}$$

$$\sigma_{32} = 2\mu\epsilon_{32} = \mu(\frac{\partial u_3}{\partial x_2} + \frac{\partial u_2}{\partial x_3}) \tag{5.32}$$

$$\sigma_{33} = K\epsilon_{kk} + 2\mu(\epsilon_{33} - \frac{1}{3}\epsilon_{kk}) = (K - \frac{2}{3}\mu)(\frac{\partial u_1}{\partial x_1} + \frac{\partial u_2}{\partial x_2} + \frac{\partial u_3}{\partial x_3}) + 2\mu\frac{\partial u_3}{\partial x_3} \tag{5.33}$$

Here, we also use $1, 2, 3$ to denote the components of σ in x, y, z directions to simplify the notation. Then we can rewrite governing equations in terms of the x, y, z components.

$$\begin{aligned}
\rho\frac{\partial^2 u_1}{\partial t^2} &= \frac{\partial \sigma_{11}}{\partial x_1} + \frac{\partial \sigma_{12}}{\partial x_2} + \frac{\partial \sigma_{13}}{\partial x_3} \\
&= (K - \frac{2}{3}\mu)(\frac{\partial^2 u_1}{\partial x_1^2} + \frac{\partial^2 u_2}{\partial x_1 \partial x_2} + \frac{\partial^2 u_3}{\partial x_1 \partial x_3}) + 2\mu\frac{\partial^2 u_1}{\partial x_1^2} \\
&+ \mu(\frac{\partial^2 u_1}{\partial x_2^2} + \frac{\partial^2 u_2}{\partial x_2 \partial x_1}) + \mu(\frac{\partial^2 u_1}{\partial x_3^2} + \frac{\partial^2 u_3}{\partial x_3 \partial x_1})
\end{aligned} \tag{5.34}$$

$$\begin{aligned}
\rho\frac{\partial^2 u_2}{\partial t^2} &= \frac{\partial \sigma_{21}}{\partial x_1} + \frac{\partial \sigma_{22}}{\partial x_2} + \frac{\partial \sigma_{23}}{\partial x_3} \\
&= \mu(\frac{\partial^2 u_2}{\partial x_1^2} + \frac{\partial^2 u_1}{\partial x_1 \partial x_2}) + (K - \frac{2}{3}\mu)(\frac{\partial^2 u_1}{\partial x_2 \partial x_1} + \frac{\partial^2 u_2}{\partial x_2^2} + \frac{\partial^2 u_3}{\partial x_2 \partial x_3}) \\
&+ 2\mu\frac{\partial^2 u_2}{\partial x_2^2} + \mu(\frac{\partial^2 u_2}{\partial x_3^2} + \frac{\partial^2 u_3}{\partial x_3 \partial x_2})
\end{aligned} \tag{5.35}$$

$$\begin{aligned}
\rho\frac{\partial^2 u_3}{\partial t^2} &= \frac{\partial \sigma_{31}}{\partial x_1} + \frac{\partial \sigma_{32}}{\partial x_2} + \frac{\partial \sigma_{33}}{\partial x_3} \\
&= \mu(\frac{\partial^2 u_3}{\partial x_1^2} + \frac{\partial^2 u_1}{\partial x_1 \partial x_3}) + \mu(\frac{\partial^2 u_3}{\partial x_2^2} + \frac{\partial^2 u_2}{\partial x_2 \partial x_3}) \\
&+ (K - \frac{2}{3}\mu)(\frac{\partial^2 u_1}{\partial x_3 \partial x_1} + \frac{\partial^2 u_2}{\partial x_3 \partial x_2} + \frac{\partial^2 u_3}{\partial x_3^2}) + 2\mu\frac{\partial^2 u_3}{\partial x_3^2}
\end{aligned} \tag{5.36}$$

We can use them to impose the SAT terms for displacements in x, y, z directions using formulations from Almquist and Dunham (2021). For Neuman boundary conditions, we have

$$\begin{aligned}
SAT_1 = &- H^{-1}[e_3 H_3(e_3^T[T_{11}^3 u_1 + T_{12}^3 u_2 + T_{13}^3 u_3] - g_1^3)] \\
&- H^{-1}[e_4 H_4(e_4^T[T_{11}^4 u_1 + T_{12}^4 u_2 + T_{13}^4 u_3] - g_1^4)] \\
&- H^{-1}[e_5 H_5(e_5^T[T_{11}^5 u_1 + T_{12}^5 u_2 + T_{13}^5 u_3] - g_1^5)] \\
&- H^{-1}[e_6 H_6(e_6^T[T_{11}^6 u_1 + T_{12}^6 u_2 + T_{13}^6 u_3] - g_1^6)] \qquad (5.37) \\
SAT_2 = &- H^{-1}[e_3 H_3(e_3^T[T_{21}^3 u_1 + T_{22}^3 u_2 + T_{23}^3 u_3] - g_2^3)] \\
&- H^{-1}[e_4 H_4(e_4^T[T_{21}^4 u_1 + T_{22}^4 u_2 + T_{23}^4 u_3] - g_2^4)] \\
&- H^{-1}[e_5 H_5(e_5^T[T_{21}^5 u_1 + T_{22}^5 u_2 + T_{23}^5 u_3] - g_2^5)] \\
&- H^{-1}[e_6 H_6(e_6^T[T_{21}^6 u_1 + T_{22}^6 u_2 + T_{23}^6 u_3] - g_2^6)] \qquad (5.38) \\
SAT_3 = &- H^{-1}[e_3 H_3(e_3^T[T_{31}^3 u_1 + T_{32}^3 u_2 + T_{33}^3 u_3] - g_3^3)] \\
&- H^{-1}[e_4 H_4(e_4^T[T_{31}^4 u_1 + T_{32}^4 u_2 + T_{33}^4 u_3] - g_3^4)] \\
&- H^{-1}[e^5 H_5(e_5^T[T_{31}^5 u_1 + T_{32}^5 u_2 + T_{33}^5 u_3] - g_3^5)] \\
&- H^{-1}[e^6 H_6(e_6^T[T_{32}^6 u_1 + T_{32}^6 u_2 + T_{33}^6 u_3] - g_3^6)] \qquad (5.39)
\end{aligned}$$

For Dirichlet conditions, we have

$$\begin{aligned}
S\tilde{A}T_1 = {} & H^{-1}(T_{11}^1 - Z_{11}^1)^T e_1 H_1 (e_1^T u_1 - g_1^1) \\
& + H^{-1}(T_{21}^1 - Z_{21}^1)^T e_1 H_1 (e_1^T u_2 - g_2^1) \\
& + H^{-1}(T_{31}^1 - Z_{31}^1)^T e_1 H_1 (e_1^T u_3 - g_3^1) \\
& + H^{-1}(T_{11}^2 - Z_{11}^2)^T e_2 H_2 (e_2^T u_1 - g_1^2) \\
& + H^{-1}(T_{21}^2 - Z_{21}^2)^T e_2 H_2 (e_2^T u_2 - g_2^2) \\
& + H^{-1}(T_{31}^2 - Z_{31}^2)^T e_2 H_2 (e_2^T u_3 - g_3^2) \qquad (5.40) \\
S\tilde{A}T_2 = {} & H^{-1}(T_{12}^1 - Z_{12}^1)^T e_1 H_1 (e_1^T u_1 - g_1^1) \\
& + H^{-1}(T_{22}^1 - Z_{22}^1)^T e_1 H_1 (e_1^T u_2 - g_2^1) \\
& + H^{-1}(T_{32}^1 - Z_{32}^1)^T e_1 H_1 (e_1^T u_3 - g_3^1) \\
& + H^{-1}(T_{12}^2 - Z_{12}^2)^T e_2 H_2 (e_2^T u_1 - g_1^2) \\
& + H^{-1}(T_{22}^2 - Z_{22}^2)^T e_2 H_2 (e_2^T u_2 - g_2^2) \\
& + H^{-1}(T_{32}^2 - Z_{32}^2)^T e_2 H_2 (e_2^T u_3 - g_3^2) \qquad (5.41) \\
S\tilde{A}T_3 = {} & H^{-1}(T_{13}^1 - Z_{13}^1)^T e_1 H_1 (e_1^T u_1 - g_1^1) \\
& + H^{-1}(T_{23}^1 - Z_{23}^1)^T e_1 H_1 (e_1^T u_2 - g_2^1) \\
& + H^{-1}(T_{33}^1 - Z_{33}^1)^T e_1 H_1 (e_1^T u_3 - g_3^1) \\
& + H^{-1}(T_{13}^2 - Z_{13}^2)^T e_2 H_2 (e_2^T u_1 - g_1^2) \\
& + H^{-1}(T_{23}^2 - Z_{23}^2)^T e_2 H_2 (e_2^T u_2 - g_2^2) \\
& + H^{-1}(T_{33}^2 - Z_{33}^2)^T e_2 H_2 (e_3^T u_3 - g_3^2) \qquad (5.42)
\end{aligned}$$

5.3.4 Results. We discretize the problem on a truncated 128km $\times$ 128km $\times$ 128 km domain. The simulation parameters are chosen to be the values in Jiang and Erickson (2020). We export results on stations on the fault according to the problem description. We run our simulations for 1800 years and compare our results with results from other researchers on similar problems.

We compare our results using BEMs from Cattania's group. This method only requires solving problems on the fault surface, and does not require domain truncation. Previous results have shown that domain truncation in volume-based methods would affect earthquake cycles

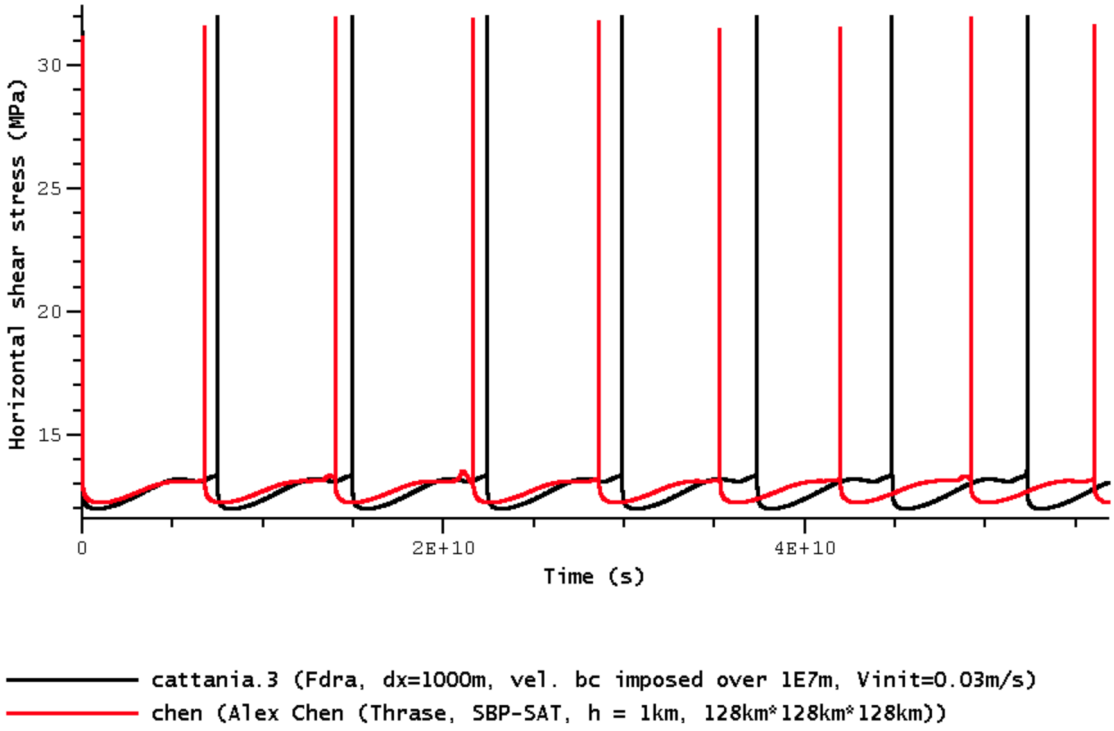

Figure 21. Comparison of shear stress along slip directions

We first look at the changes in shear stress along the slip direction for a sample location on the fault. We compare our results with Cattania's group using boundary element methods and plot the result in Figure 21. We see similar ranges of state variables and similar behaviors of jumps in state variables during the transition between aseismic slips and seismic slips. We then compare the slip for the same location on the fault and plot them in Figure 22.

Preliminary results have shown agreement in the modeling of the same problems using different methods. Current results from other simulations are mainly based on boundary element methods, which take hours to run. Our methods are volume-based and have more than 100 times higher degrees of freedom. With the GPU-accelerated MGCG as a solver, our simulation time is cut to around 8 hours, with around $0.2s$ for each round of solving linear system and updating values. This is down from weeks and months of estimated time using direct methods if we have sufficient large enough memory for matrix factorization.

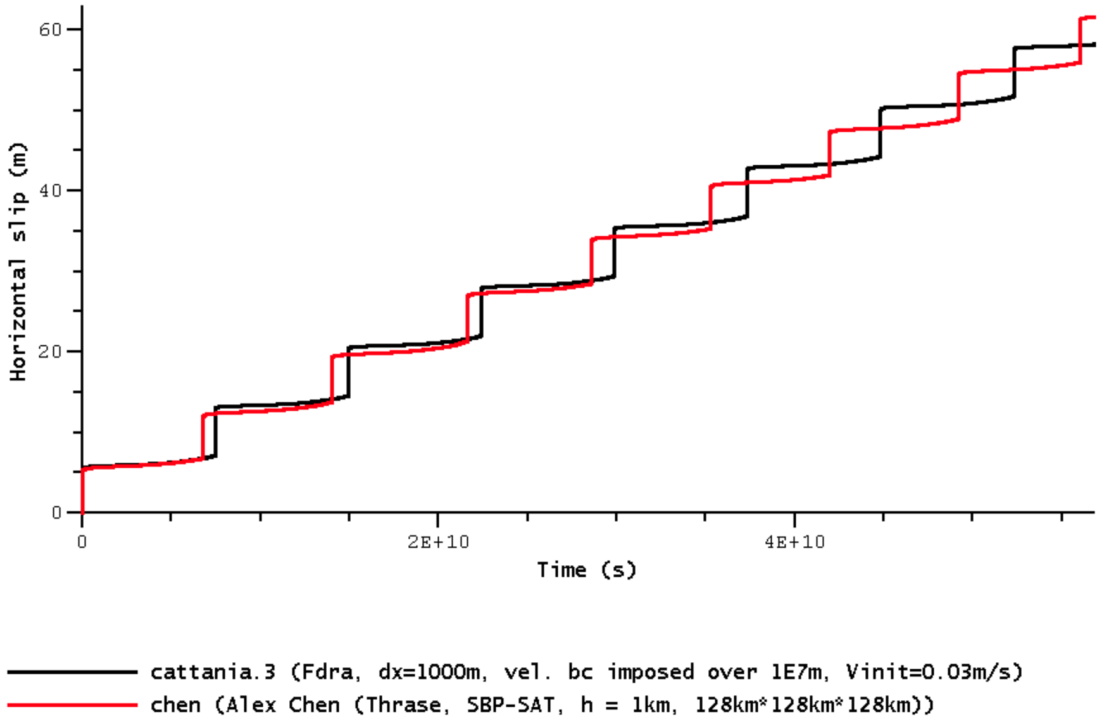

Figure 22. Comparison of shear stress along slip directions

CHAPTER VI

CONCLUSION

6.1 Conclusion

In this thesis, I present the research work during my PhD to apply HPC methods to earthquake cycle simulations. In particular, I focus on designing efficient iterative solvers for the numerical stable SBP-SAT finite difference methods. There are mainly two novel contributions of the work. Firstly, I design matrix-free GPU kernels for SBP-SAT methods based on traditional stencil computation. Secondly, I designed a geometric multigrid preconditioner that is compatible with my matrix-free GPU kernels based on the existing work of the SBP-preserving interpolation operators. The combined approach has been proven efficient in solving linear systems formed with the SBP-SAT methods, and we are outperforming the state-of-the-art implementations from well-known scientific computing libraries and proprietary software. We then apply this approach to solving SEAS modeling problems and achieve more than 100x speedup compared to traditional methods. More importantly, this approach is memory efficient that allows us to solve a 3D simulation problem formulated with the SBP-SAT method that can not be solved by factorization-based direct methods. The work presented in this thesis is valuable not only to earth science research but also to numerous other scientific fields where SBP-SAT methods can be applied.

6.2 Future Work

This work is focused on the second-order SBP-SAT methods. SBP-SAT methods are known to be high order accuracy, and using higher-order SBP operators will increase arithmetic intensity that will increase the performance of matrix-free GPU kernels compared to SpMV operators.

In addition, the matrix-free kernels presented in this paper are only implemented on a single GPU. Although this is enough to solve the problems presented in this paper, it would be more important in the HPC aspect to design matrix-free kernels that can run on multiple GPUs across different nodes. ParallelStencils.jl is a Julia package that enables large-scale stencil-based computations using built-in modules that utilize MPI for communication. It's also built on top of KernelAbstractions.jl, a Julia package that targets heterogeneous platforms. Our code on matrix-

free SBP-SAT methods can be developed with ParallelStencils.jl to run on supercomputers built with CPUs/GPUs from different vendors.

In this work, we only apply Richardson iteration as matrix-free smoothers for our multigrid preconditioners. During our research, we observed faster convergence with higher-order Krylov subspace methods as smoother. These methods are also highly suitable for GPU architectures and can be implemented matrix-free. In future work, we can explore using higher order second-order Richardson methods and Chebyshev iterations as smoothers in multigrid methods to further reduce steps till convergence for CG.

REFERENCES CITED

Abdelfattah, A., Anzt, H., Boman, E. G., Carson, E., Cojean, T., Dongarra, J., ... others (2021). A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *The International Journal of High Performance Computing Applications*, *35*(4), 344–369.

ADELI, H., & VISHNUBHOTLA, P. (1987). Parallel processing. *Computer-Aided Civil and Infrastructure Engineering*, *2*(3), 257–269.

Adler, J. H., Benson, T. R., Cyr, E. C., MacLachlan, S. P., & Tuminaro, R. S. (2016). Monolithic multigrid methods for two-dimensional resistive magnetohydrodynamics. *SIAM Journal on Scientific Computing*, *38*(1), B1–B24.

Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., ... Tomov, S. (2009). Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of physics: Conference series* (Vol. 180, p. 012037).

Almquist, M., & Dunham, E. M. (2021). Elastic wave propagation in anisotropic solids using energy-stable finite differences with weakly enforced boundary and interface conditions. *Journal of Computational Physics*, *424*, 109842. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0021999120306161` doi: https://doi.org/10.1016/j.jcp.2020.109842

Ashby, S. F., Manteuffel, T. A., & Otto, J. S. (1992). A comparison of adaptive chebyshev and least squares polynomial preconditioning for hermitian positive definite linear systems. *SIAM Journal on Scientific and Statistical Computing*, *13*(1), 1–29.

Avouac, J.-P. (2015). From geodetic imaging of seismic and aseismic fault slip to dynamic modeling of the seismic cycle [Journal Article]. *Annual Review of Earth and Planetary Sciences*, *43*(Volume 43, 2015), 233-271. Retrieved from `https://www.annualreviews.org/content/journals/10.1146/annurev-earth-060614-105302` doi: https://doi.org/10.1146/annurev-earth-060614-105302

Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., ... Zhang, J. (2023). *PETSc Web page.* `https://petsc.org/`. Retrieved from `https://petsc.org/`

Barker, B. (2015). Message passing interface (mpi). In *Workshop: High performance computing on stampede* (Vol. 262).

Baskaran, M. M., & Bordawekar, R. (2009). Optimizing sparse matrix-vector multiplication on gpus. *IBM research report RC24704*(W0812–047).

Bathe, K.-J. (2006). *Finite element procedures.* Klaus-Jurgen Bathe.

Beckingsale, D. A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., ... Scogland, T. R. (2019). RAJA: Portable performance for large-scale scientific applications. In *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)* (p. 71-81). doi: 10.1109/P3HPC49587.2019.00012

Bell, N., & Garland, M. (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis* (p. 1-11). doi: 10.1145/1654059.1654078

Bell, N., Olson, L. N., & Schroder, J. (2022). Pyamg: algebraic multigrid solvers in python. *Journal of Open Source Software*, *7*(72), 4142.

Besard, T., Foket, C., & De Sutter, B. (2018). Effective extensible programming: unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, *30*(4), 827–841. doi: 10.1109/TPDS.2018.2872064

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM review*, *59*(1), 65–98. doi: 10.1137/141000671

Boué, P., Denolle, M., Hirata, N., Nakagawa, S., & Beroza, G. C. (2016, 06). Beyond basin resonance: characterizing wave propagation using a dense array and the ambient seismic field. *Geophysical Journal International*, *206*(2), 1261-1272. doi: 10.1093/gji/ggw205

Brandt, A. (1986). Algebraic multigrid theory: The symmetric case. *Applied mathematics and computation*, *19*(1-4), 23–56.

Brandt, A. (2006). Guide to multigrid development. In *Multigrid methods: Proceedings of the conference held at köln-porz, november 23–27, 1981* (pp. 220–312).

Briggs, W. L., Henson, V. E., & McCormick, S. F. (2000a). *A multigrid tutorial*. SIAM.

Briggs, W. L., Henson, V. E., & McCormick, S. F. (2000b). *A multigrid tutorial (2nd ed.)*. USA: Society for Industrial and Applied Mathematics.

Briggs, W. L., Henson, V. E., & McCormick, S. F. (2000c). *A multigrid tutorial, second edition* (Second ed.). Society for Industrial and Applied Mathematics.

Carpenter, M. H., Gottlieb, D., & Abarbanel, S. (1994). Time-stable boundary conditions for finite-difference schemes solving hyperbolic systems: Methodology and application to high-order compact schemes. *Journal of Computational Physics*, *111*(2), 220–236. doi: 10.1006/jcph.1994.1057

Carter Edwards, H., Trott, C. R., & Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, *74*(12), 3202-3216.

Chtchelkanova, A., Gunnels, J., Morrow, G., Overfelt, J., & Van De Geijn, R. A. (1997). Parallel implementation of blas: General techniques for level 3 blas. *Concurrency: Practice and Experience*, *9*(9), 837–857.

Churavy, V., Aluthge, D., Smirnov, A., Schloss, J., Samaroo, J., Wilcox, L. C., ... Haraldsson, P. (2024, March). *Juliagpu/kernelabstractions.jl: v0.9.18.* Zenodo. Retrieved from `https://doi.org/10.5281/zenodo.10780635` doi: 10.5281/zenodo.10780635

Dalton, S., Bell, N., Olson, L., & Garland, M. (2014). Cusp: Generic parallel algorithms for sparse matrix and graph computations. *Version 0.5. 0*.

Del Rey Fernández, D. C., Hicken, J. E., & Zingg, D. W. (2014). Review of summation-by-parts operators with simultaneous approximation terms for the numerical solution of partial differential equations. *Computers & Fluids*, *95*, 171-196.

Dieterich, J. H. (1979a). Modeling of rock friction: 1. experimental results and constitutive equations. *Journal of Geophysical Research: Solid Earth*, *84*(B5), 2161-2168. Retrieved from `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/JB084iB05p02161` doi: https://doi.org/10.1029/JB084iB05p02161

Dieterich, J. H. (1979b). Modeling of rock friction: 2. simulation of preseismic slip. *Journal of Geophysical Research: Solid Earth*, *84*(B5), 2169-2175. Retrieved from `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/JB084iB05p02169` doi: https://doi.org/10.1029/JB084iB05p02169

Ding, N., & Williams, S. (2019). *An instruction roofline model for gpus*. IEEE.

Dongarra, J. J., Du Croz, J., Hammarling, S., & Duff, I. S. (1990, mar). A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, *16*(1), 1–17. Retrieved from `https://doi.org/10.1145/77626.79170` doi: 10.1145/77626.79170

Dongarraxz, J., Lumsdaine, A., Niu, X., Pozoz, R., & Remingtonx, K. (1994). A sparse matrix library in c++ for high performance architectures. In *Second object oriented numerics conference* (pp. 214–218).

Erickson, B., & Jiang, J. (2018). Seas benchmark problem bp1. *Retrieved from*, *1168*.

Erickson, B. A., & Day, S. M. (2016). Bimaterial effects in an earthquake cycle model using rate-and-state friction. *Journal of Geophysical Research: Solid Earth*, *121*, 2480–2506. doi: 10.1002/2015JB012470

Erickson, B. A., & Dunham, E. M. (2014). An efficient numerical method for earthquake cycles in heterogeneous media: Alternating subbasin and surface-rupturing events on faults crossing a sedimentary basin. *Journal of Geophysical Research: Solid Earth*, *119*(4), 3290–3316. doi: 10.1002/2013JB010614

Erickson, B. A., Jiang, J., Barall, M., Lapusta, N., Dunham, E. M., Harris, R., ... others (2020a). The community code verification exercise for simulating sequences of earthquakes and aseismic slip (seas). *Seismological Research Letters*, *91*(2A), 874–890.

Erickson, B. A., Jiang, J., Barall, M., Lapusta, N., Dunham, E. M., Harris, R., ... Wei, M. (2020b, 01). The Community Code Verification Exercise for Simulating Sequences of Earthquakes and Aseismic Slip (SEAS). *Seismological Research Letters*, *91*(2A), 874-890. Retrieved from `https://doi.org/10.1785/0220190248` doi: 10.1785/0220190248

Erickson, B. A., Jiang, J., Lambert, V., Barbot, S. D., Abdelmeguid, M., Almquist, M., ... others (2023). Incorporating full elastodynamic effects and dipping fault geometries in community code verification exercises for simulations of earthquake sequences and aseismic slip (seas). *Bulletin of the Seismological Society of America*, *113*(2), 499–523.

Erickson, B. A., Kozdon, J. E., & Harvey, T. (2022). A non-stiff summation-by-parts finite difference method for the scalar wave equation in second order form: Characteristic boundary conditions and nonlinear interfaces. *Journal of Scientific Computing*, *93*(1), 17.

Falgout, R. D., Jones, J. E., & Yang, U. M. (2006a, jan). Conceptual interfaces in hypre. *Future Gener. Comput. Syst.*, *22*(1), 239–251.

Falgout, R. D., Jones, J. E., & Yang, U. M. (2006b). The design and implementation of hypre, a library of parallel high performance preconditioners.. Retrieved from `https://api.semanticscholar.org/CorpusID:3237430`

Fedorenko, R. P. (1973). Iterative methods for elliptic difference equations. *Russian Mathematical Surveys*, *28*(2), 129–195.

Freeman, T. L., & Phillips, C. (1992). *Parallel numerical algorithms*. Prentice-Hall, Inc.

Gee, M. W., Siefert, C. M., Hu, J. J., Tuminaro, R. S., & Sala, M. G. (2006). *Ml 5.0 smoothed aggregation user's guide* (Tech. Rep.). Technical Report SAND2006-2649, Sandia National Laboratories.

Golub, G. H., & Varga, R. S. (1961). Chebyshev semi-iterative methods, successive overrelaxation iterative methods, and second order Richardson iterative methods. *Numerische Mathematik*, *3*(1), 157–168.

Grossmann, C. (1994). Hackbusch, w., iterative lösung großer schwach besetzter gleichungssysteme. stuttgart, b. g. teubner 1991. 382 s., dm 42,- isbn 3-519-02372-5 (leitfaden der angewandten mathematik und mechanik 69, teubern-studienbücher: Mathematik). *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, *74*(2), 96-96. Retrieved from `https://onlinelibrary.wiley.com/doi/abs/10.1002/zamm.19940740205` doi: https://doi.org/10.1002/zamm.19940740205

Gustafsson, B., Kreiss, H.-O., & Oliger, J. (1995). *Time dependent problems and difference methods* (Vol. 24). John Wiley & Sons.

Gutknecht, M. H., & Röllin, S. (2002). The Chebyshev iteration revisited. *Parallel Computing*, *28*(2), 263–283.

Hackbusch, W. (2013a). *Iterative lösung großer schwachbesetzter gleichungssysteme* (Vol. 69). Springer-Verlag.

Hackbusch, W. (2013b). *Multi-grid methods and applications* (Vol. 4). Springer Science & Business Media.

Häfner, S., Eckardt, S., Luther, T., & Könke, C. (2006). Mesoscale modeling of concrete: Geometry and numerics. *Computers & structures*, *84*(7), 450–461.

Harris, M., Owens, J., Sengupta, S., Zhang, Y., & Davidson, A. (2007). *Cudpp: Cuda data parallel primitives library.*

Hestenes, M. R., Stiefel, E., et al. (1952). *Methods of conjugate gradients for solving linear systems* (Vol. 49) (No. 1). NBS Washington, DC.

Jiang, J., & Erickson, B. (2020). Seas benchmark problems bp5-qd and bp5-fd.

Kozdon, J. E., Dunham, E. M., & Nordström, J. (2012). Interaction of waves with frictional interfaces using summation-by-parts difference operators: Weak enforcement of nonlinear boundary conditions. *Journal of Scientific Computing*, *50*, 341-367. doi: 10.1007/s10915-011-9485-3

Kozdon, J. E., Erickson, B. A., & Wilcox, L. C. (2020). Hybridized summation-by-parts finite difference methods. *Journal of Scientific Computing*. doi: 10.1007/s10915-021-01448-5

Kreiss, H., & Scherer, G. (1974). Finite element and finite difference methods for hyperbolic partial differential equations. In *Mathematical aspects of finite elements in partial differential equations; proceedings of the symposium* (pp. 195–212). Madison, WI. doi: 10.1016/b978-0-12-208350-1.50012-1

Krotkiewski, M., & Dabrowski, M. (2013). Efficient 3D stencil computations using CUDA. *Parallel Computing*, *39*(10), 533–548.

Krylov, A. N. (1931). On the numerical solution of the equation by which in technical questions frequencies of small oscillations of material systems are determined. *Izvestija AN SSSR (News of Academy of Sciences of the USSR), Otdel. mat. i estest. nauk*, *7*(4), 491–539.

Li, K., Yang, W., & Li, K. (2014). Performance analysis and optimization for spmv on gpu using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems*, *26*(1), 196–205.

Liu, C., & Henshaw, W. (2023). Multigrid with nonstandard coarse-level operators and coarsening factors. *Journal of Scientific Computing*, *94*(3), 58.

Lotto, G. C., & Dunham, E. M. (2015). High-order finite difference modeling of tsunami generation in a compressible ocean from offshore earthquakes. *Computational Geosciences*, *19*(2), 327–340. doi: 10.1007/s10596-015-9472-0

Lubarda, M., & Lubarda, V. (2019). *Intermediate solid mechanics.* Cambridge University Press. doi: 10.1017/9781108589000.011

Mandel, J. (1988). Algebraic study of multigrid methods for symmetric, definite problems. *Applied mathematics and computation*, *25*(1), 39–56.

Marone, C. (1998). Laboratory-derived friction laws and their application to seismic faulting [Journal Article]. *Annual Review of Earth and Planetary Sciences*, *26*(Volume 26, 1998), 643-696. Retrieved from `https://www.annualreviews.org/content/journals/10.1146/annurev.earth.26.1.643` doi: https://doi.org/10.1146/annurev.earth.26.1.643

Marshall, S. T., Funning, G. J., Krueger, H. E., Owen, S. E., & Loveless, J. P. (2017). Mechanical models favor a ramp geometry for the ventura-pitas point fault, california. *Geophysical Research Letters*, *44*(3), 1311-1319. Retrieved from `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2016GL072289` doi: https://doi.org/10.1002/2016GL072289

Mattsson, K. (2003, Feb 01). Boundary procedures for summation-by-parts operators. *Journal of Scientific Computing*, *18*(1), 133-153.

Mattsson, K., Ham, F., & Iaccarino, G. (2009). Stable boundary treatment for the wave equation on second-order form. *Journal of Scientific Computing*, *41*(3), 366–383. doi: 10.1007/s10915-009-9305-1

Mattsson, K., & Nordström, J. (2004). Summation by parts operators for finite difference approximations of second derivatives. *Journal of Computational Physics*, *199*(2), 503–540. doi: 10.1016/j.jcp.2004.03.001

McCormick, S., & Ruge, J. (1982). Multigrid methods for variational problems. *SIAM Journal on Numerical Analysis*, *19*(5), 924–929.

Medina, DS and St-Cyr, A. and Warburton, T. (2014). OCCA: A unified approach to multithreading languages. *arXiv preprint arXiv:1403.0968*.

Mohammadi, M. S., Cheshmi, K., Gopalakrishnan, G., Hall, M., Dehnavi, M. M., Venkat, A., ... Strout, M. M. (2018). Sparse matrix code dependence analysis simplification at compile time. *arXiv preprint arXiv:1807.10852*.

Nordström, J., & Eriksson, S. (2010). Fluid structure interaction problems: The necessity of a well posed, stable and accurate formulation. *Communications in Computational Physics*, *8*(5), 1111–1138. doi: https://doi.org/10.4208/cicp.260409.120210a

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). Gpu computing. *Proceedings of the IEEE*, *96*(5), 879–899.

Petersson, N. A., & Sjögreen, B. (2012). Stable and efficient modeling of anelastic attenuation in seismic wave propagation. *Communications in Computational Physics*, *12*(1), 193–225. doi: 10.4208/cicp.201010.090611a

Ranocha, H., & Nordström, J. (2021). A new class of a stable summation by parts time integration schemes with strong initial conditions. *Journal of Scientific Computing*, *87*(1), 1–25.

Roten, D., Cui, Y., Olsen, K. B., Day, S. M., Withers, K., Savran, W. H., ... Mu, D. (2016). High-frequency nonlinear earthquake simulations on petascale heterogeneous supercomputers. In *Sc '16: Proceedings of the international conference for high performance computing, networking, storage and analysis* (p. 957-968). doi: 10.1109/SC.2016.81

Ruge, J. W., & Stüben, K. (1987). Algebraic multigrid. In *Multigrid methods* (pp. 73–130). SIAM.

Ruggiu, A. A., Weinerfelt, P., & Nordström, J. (2018). A new multigrid formulation for high order finite difference methods on summation-by-parts form. *Journal of Computational Physics*, *359*, 216–238.

Ruggiu, A. A., Weinerfelt, P., & Nordström, J. (2018). A new multigrid formulation for high order finite difference methods on summation-by-parts form. *Journal of Computational Physics*, *359*, 216-238. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0021999118300214` doi: https://doi.org/10.1016/j.jcp.2018.01.011

Ruggiu, A. A., Weinerfelt, P., & Norström, J. (2018). A new multigrid formulation for high order finite difference methods on summation-by-parts form. *Journal of Computational Physics*, *359*, 216–238. doi: https://doi.org/10.1016/j.jcp.2018.01.011

Ruina, A. (1983). Slip instability and state variable friction laws. *Journal of Geophysical Research: Solid Earth*, *88*(B12), 10359-10370. Retrieved from `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/JB088iB12p10359` doi: https://doi.org/10.1029/JB088iB12p10359

Saad, Y. (2003). *Iterative methods for sparse linear systems* (Second ed.). Society for Industrial and Applied Mathematics. Retrieved from `https://epubs.siam.org/doi/abs/10.1137/1.9780898718003` doi: 10.1137/1.9780898718003

Smailbegovic, F., Gaydadjiev, G. N., & Vassiliadis, S. (2005). Sparse matrix storage format. In *Proceedings of the 16th annual workshop on circuits, systems and signal processing* (pp. 445–448).

Sridhar, A., Tissaoui, Y., Marras, S., Shen, Z., Kawczynski, C., Byrne, S., ... Schneider, T. (2022). Large-eddy simulations with ClimateMachine v0.2.0: a new open-source code for atmospheric simulations on GPUs and CPUs. *Geoscientific Model Development*, *15*(15), 6259–6284.

Stanimirovic, I. P., & Tasic, M. B. (2009). Performance comparison of storage formats for sparse matrices. *Ser. Mathematics and Informatics*, *24*(1), 39–51.

Stolk, C. C., Ahmed, M., & Bhowmik, S. K. (2014). A multigrid method for the helmholtz equation with optimized coarse grid corrections. *SIAM Journal on Scientific Computing*, *36*(6), A2819–A2841.

Strand, B. (1994). Summation by parts for finite difference approximations for d/dx. *Journal of Computational Physics*, *110*(1), 47–67. doi: 10.1006/jcph.1994.1005

Stüben, K. (2001). A review of algebraic multigrid. *Numerical Analysis: Historical Developments in the 20th Century*, 331–359.

Sundar, H., Stadler, G., & Biros, G. (2015). Comparison of multigrid algorithms for high-order continuous finite element discretizations. *Numerical Linear Algebra with Applications*, *22*(4), 664–680.

Svärd, M., & Nordström, J. (2014). Review of summation-by-parts schemes for initial-boundary-value problems. *Journal of Computational Physics*, *268*, 17-38. doi: https://doi.org/10.1016/j.jcp.2014.02.031

Swim, E., Benra, F.-K., Dohmen, H. J., Pei, J., Schuster, S., & Wan, B. (2011). A comparison of one-way and two-way coupling methods for numerical analysis of fluid-structure interactions. *Journal of Applied Mathematics*, *2011*, 1–16. doi: 10.1155/2011/853560

Tatebe, O. (1993). The multigrid preconditioned conjugate gradient method. In *Nasa conference publication* (pp. 621–621).

Trottenberg, U., Oosterlee, C. W., & Schuller, A. (2000). *Multigrid*. Elsevier.

Tsitouras, C. (2011). Runge–kutta pairs of order 5(4) satisfying only the first column simplifying assumption. *Computers & Mathematics with Applications*, *62*(2), 770-775. Retrieved from `https://www.sciencedirect.com/science/article/pii/S0898122111004706` doi: https://doi.org/10.1016/j.camwa.2011.06.002

Tullis, T. E., Richards-Dinger, K., Barall, M., Dieterich, J. H., Field, E. H., Heien, E. M., ... Yikilmaz, M. B. (2012, 11). Generic Earthquake Simulator. *Seismological Research Letters*, *83*(6), 959-963. Retrieved from `https://doi.org/10.1785/0220120093` doi: 10.1785/0220120093

Vaněk, P., Mandel, J., & Brezina, M. (1996). Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, *56*(3), 179–196.

Vizitiu, A., Itu, L., Niţă, C., & Suciu, C. (2014). Optimized three-dimensional stencil computation on Fermi and Kepler GPUs. In *2014 ieee high performance extreme computing conference (hpec)* (pp. 1–6).

Wait, R., & Brown, N. (1988). Overlapping block methods for solving tridiagonal systems on transputer arrays. *Parallel Computing*, *8*(1), 325-333. Retrieved from `https://www.sciencedirect.com/science/article/pii/0167819188901366` (Proceedings of the International Conference on Vector and Parallel Processors in Computational Science III) doi: https://doi.org/10.1016/0167-8191(88)90136-6

Wesseling, P. (2004). An introduction to multigrid methods. rt edwards. *Inc.,*, *2*, 2.

Wilt, N. (2013). *The cuda handbook: A comprehensive guide to gpu programming*. Pearson Education.

Xu, J., & Zikatanov, L. (2017). Algebraic multigrid methods. *Acta Numerica*, *26*, 591–721.

Yan, S., Li, C., Zhang, Y., & Zhou, H. (2014). YaSpMV: Yet another SpMV framework on GPUs. In *Proceedings of the 19th acm sigplan symposium on principles and practice of parallel programming* (p. 107–118). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2555243.2555255

Yang, U. M., et al. (2002). BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, *41*(1), 155–177.

Ying, W., & Henriquez, C. (2007). Hybrid finite element method for describing the electrical response of biological cells to applied fields. *IEEE Transactions on Bio-medical Engineering*, *54*(4), 611–620. doi: 10.1109/TBME.2006.889172

Young, D. M. (2014). *Iterative solution of large linear systems*. Elsevier.