Noname manuscript No.
(will be inserted by the editor)

A Performance Analysis of a Hybridized SBP-SAT Finite-Difference Method for Large Scale Earth Science Applications

Alexandre Chen · Brittany A. Erickson · Jeremy E. Kozdon

Received: date / Accepted: date

Abstract We present performance results from a new hybridized finite difference method for the spatial discretization of partial differential equations. The method is based on the standard Summation-By-Parts method with weak enforcement of boundary and interface conditions through the Simultaneous-Approximation-Term. We analyze the performance when applying the hybrid method to Poisson's equation which arises in many steady-state physical problems, focusing on an application in Earth science. When solving the resulting linear system we compare direct and iterative solvers on both CPU and GPU, evaluating the performance on meshes with different numbers of computational blocks. Our results demonstrate the advantages of using the hybrid method in solving large-scale problems under the restriction of system resources by utilizing techniques from parallel computing.

1 Introduction and Background

1.1 Poisson's Equation and an Application in Earth Science

Poisson's equation is a partial differential equation (PDE) of elliptic type that is widely used in physics, fluid dynamics, mechanical engineering, and other fields to study steady-state problems. The equation is given by

$$\nabla^2 \varphi = f, \tag{1}$$

Alexandre Chen and B. A. Erickson
Computer and Information Science
1202 University of Oregon
1477 E. 13th Ave.
Eugene, OR 97403–1202
E-mail: {chern,bae}@cs.uoregon.edu

J. E. Kozdon
Department of Applied Mathematics,
Naval Postgraduate School,
833 Dyer Road,
Monterey, CA 93943–5216
E-mail: jekozdon@nps.edu

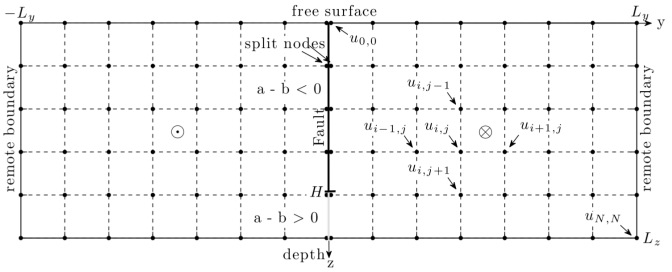

Fig. 1: A 2D simplified model in an earthquake simulation. Figure from Erickson and Dunham (2014)

where f and ϕ are real or complex-valued functions on a Euclidean space, with f a given source function and φ is sought. When $f = 0$, we obtain Laplace's Equation.

In computational seismology, Poisson's equation arises when describing the 2D antiplane problem as shown in figure 1, where the out-of-plane displacement $u = \phi$ is sought (Erickson and Dunham, 2014). As illustrated in the figure, a section in the $-yz$ plane is considered, containing symmetry with respect to the z axis. An 1D earthquake fault (an interface) is located along z-axis and is subject to a specified friction law. Tectonic motion is captured by setting the remote boundaries to be displaced at a slow plate rate of ≈ 32mm/yr, which is enforced by applying Dirichlet boundary conditions. Although the boundary conditions are changing through the time at an extremely low rate, we have assumed a quasi-static response. A sequence of earthquakes nucleate at the fault in response to the remote tectonic loading. In this earthquake cycle simulation, the fault is a thin zone of crushed rock separating blocks of the Earth's crust. When an earthquake occurs on the fault, the rock on one side of the fault is displaced with respect to the other side, and this jump in displacement across the fault is known as slip. The fault length can be of several hundreds of kilometers, with frictional properties on the order of microns, which gives rise to large problems in simulation. Earth's free surface is at $z = 0$ and we also assume a free surface at depth, corresponding to Neumann boundary conditions. Due to the symmetry of the system, the problem can be further simplified by considering only one side of the fault. Once we have the numerical solution for one side of the fault, the other side can be easily obtained from symmetry properties.

The assumption of steady-state motion in the anti-plane scenario gives rise to the following anisotropic version of Poisson's equation in a two dimensional domain Ω:

$$-\nabla \cdot (\boldsymbol{b}\nabla \boldsymbol{u}) = \boldsymbol{f}, \text{ on } \Omega \tag{2a}$$

$$u = g_D, \text{ on } \partial\Omega_D \tag{2b}$$

$$\boldsymbol{n} \cdot \boldsymbol{b}\nabla \boldsymbol{u} = g_N, \text{ on } \partial\Omega_N \tag{2c}$$

$$\begin{cases} \{\{\boldsymbol{n} \cdot \boldsymbol{b}\nabla u\}\} = 0 \\ [\![u]\!] = \delta \end{cases} \text{ on } \partial\Gamma_I, \tag{2d}$$

where the field u is the material displacement. Here, $\boldsymbol{b}(x, y)$ is a matrix valued function that is symmetric positive definite and the scalar function $\boldsymbol{f}(x, y)$ is the

source function. The boundary conditions of the domain have been partitioned into a Dirichlet one and a Neumann one, namely, $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$ and $\partial\Omega_D \cap \partial\Omega_N = \emptyset$. At the Neumann boundaries, the vector $\boldsymbol{n}$ is the outward pointing normal. We have also introduced an internal interface Γ_I, and along this interface, the $\boldsymbol{b}-$weighted normal derivative is taken to be continuous, with jumps allowed in the scalar field. Jumps allow us to apply this method to physical problems where displacements occur across an interface, e.g. earthquakes that occur along a fault which forms an interface in the solid Earth. $\{\{w\}\} = w^+ + w^-$ here denotes the sum of the scalar quantity on both sides of the interface and $[\![w]\!] = w^+ - w^-$ is the difference across the interface. For system with jumps across the interface, we define a non-zero constant δ vector to depict this discontinuity. When there is no jump across the interface, δ is set to be zero.

1.2 Discretization with the SBP-SAT Scheme and Numerical Solution Methods

We mesh our domain with rectilinear grids in order to apply finite difference methods. For real geographical domains where rectilinear meshing can not be applied, we can use coordinate transformations to transform the physical domain into a logical rectangular domain where we can apply rectilinear meshing. The solutions obtained from the logical domain can later be transformed back into solutions for the real domain using inverse transformation. For our study here, we consider the unit square for simplicity without losing generality. The detailed technique on coordinate transformation can be found in Kozdon, Erickson, et al. (2020) and won't be covered here.

Summation-By-Parts (SBP) finite difference methods have been proposed to solve problems with complex geometries such as the problem in this paper due to their desirable properties of high order accuracy and provable stability (Kreiss and Scherer, 1974; Kreiss and Scherer, 1977; Strand, 1994; Mattsson and Nordström, 2004). The inter-block coupling conditions can be enforced weakly using the Simultaneous-Approximation Term (SAT) method (Carpenter, Gottlieb, et al., 1994; Carpenter, Nordström, et al., 1999). The SAT term here is analogous to the penalty term in discrete Galerkin methods. More details on SBP-SAT method will be covered in 2.

In earthquake cycle simulations, earthquake nucleation and rupture propagation is simulated over thousands of years, where quasi-steady state problems are formed to depict slow and quiescent periods between earthquakes (Erickson and Dunham, 2014). In the steady-state regime, we need to solve elliptic partial differential equations, which will result in large linear systems of equations for realistically complex problems under the constraint of stability requirement for time stepping methods. In order to obtain stable solutions over long time scales, we need to apply a fine mesh in the spatial domain, and this is where large scale linear problems arise. This project is constructed around one key challenge: How can we obtain numerical results for a large linear system formed by SBP-SAT operators in order to study earthquake cycle simulations over long time scales?

In the terminology of algorithm, the time complexity refers to the number of steps required for an algorithm, and the space complexity refers to the total space taken by the algorithm to store the input data and intermediate results with respect to the input size. Solving a linear system of size $n \times n$ directly with an

LU decomposition or other factorization-based direct method is known to have the time complexity of $\mathcal{O}(n^3)$ and space complexity of $\mathcal{O}(n^2)$. The computational complexity mainly comes from the process of factorization which has $\mathcal{O}(n^3)$. After obtaining the factorization, using forward/backward substitution to solve the system has the time complexity of $\mathcal{O}(n^2)$. In time stepping methods, we can reuse the factorization results during each linear solve, but the space complexity of factorization has limited the size of the problem that system is capable of doing. Another common issue is that the matrices that we form using the SBP-SAT methods can be expressed in a sparse matrix fashion which reduces the cost of matrix storage and operations. But this sparsity can be destroyed during the LU factorization, and it relies on certain factorization algorithms that can preserve sparsity to obtain optimal results.

One way to get around the size restriction imposed by the space complexity of direct solvers is using iterative methods. Iterative methods convert the problem of solving a linear system into a problem in optimization. A common example is the Conjugate Gradient (CG) method which is a traditional iterative solver but more known to computer scientists in recent years because it has been widely used in machine learning to minimize the loss function defined with information entropy. CG is particularly suitable for solving a linear system that has a positive definite (PD) left-hand side. For information on other iterative solvers, refer to Saad (2003). Iterative solvers avoid the challenge of obtaining a factorization for sparse matrices by using repeated matrix-vector products to obtain an approximate numerical result, approaching the exact solution with proven asymptotic accuracy. Given that in numerical methods, the accuracy is limited by the round-off error, iterative solvers can provide a numerical result that matches the accuracy of the result obtained from a direct solver. Iterative solvers have other desirable properties in that we can gain efficiency by loosening the accuracy constraint, obtaining a less accurate result that is sufficient for the study in mind. The linear algebra operations such as matrix-vector multiplications can be easily accelerated by packages such as Basic Linear Algebra Subprograms (BLAS) on both CPU and GPU architecture. In 3, we will study different iterative implementations to compare the accuracy, stability and performance for our problem in search for an optimal iterative method for this specific problems.

Another way to bypass the limitation of using direct solvers on a large single system is to use an new hybridization technique that was proposed for SBP-SAT methods (Kozdon, Erickson, et al., 2020). This method reduces the system size by writing the numerical method in a way that leverages the Schur complement and eliminates degrees of freedom from within the element, leaving only degrees of freedom on element boundaries. We set the values on all the interfaces to be given input data known as the trace variables. These independent trace variables along the faces of the blocks are introduced so the inter-block coupling penalty terms can be expressed merely as a function of the trace variables. Hence the solution in each block is uniquely determined by these trace variables. In this hybrid method, the problem is broken into two pieces, a *local problem* and a *global problem*. *local problem* refers to the solution within the block given trace data, and the *global problem* refers to the value of the trace variable, given the block data. The *local problem* and the *global problem* are connected via a Schur complement. This is an extension of existing SBP-SAT scheme by introducing trace variables so we

can work on a domain with multiple blocks that share interfaces. Details will be covered in section 4 when we discuss performance of the hybrid method.

1.3 Implementation

In this paper we implement algorithms solving Poisson's equation with SBP-SAT method and extended hybrid SBP-SAT method in the Julia programming language with support of various open packages from linear algebra to GPU computing. Julia is a new programming language with an emphasis on scientific computing. It is designed to solve the Two-Language-Problem that many researchers encounter when developing a prototype in a high-level language such as Python or MATLAB (for efficiency in development) and then implementing the code in a low-level language such as C++ and FORTRAN for performance. Julia is a compiled language leveraging the JIT compilation for performance. It supports dynamic notation with multiple dispatch, which gives high code readability during development and high performance in code execution. At the compilation level, Julia uses LLVM, which generates an LLVM intermediate representation that can be used to work with other languages/frameworks that are also using LLVM. The extensibility from using LLVM as a compiler has been demonstrated with Julia's capability of leveraging GPU power for HPC (Besard et al., 2019).

Other nice features of Julia include metaprogramming from the legacy of the LISP language. Metaprogramming allows us to write less code by reducing repetition. Unlike other languages or frameworks that are accessed in one programming language but written in another programming language, many Julia packages are written in Julia itself with core source code open-sourced. This makes cooperation in Julia much more handy, and for this reason, the Julia language has become one of the most fast-growing languages with professional active users in computational science who help form a booming Julia ecosystem. However, being a relatively new open language also means there is lack of official support when it comes to bugs. Our implementations were limited by the compatibility issues of different packages that have to wait for the update from independent developers who wrote these packages.

This rest of the paper is organized as follows: In Section 2, we provide a detailed description of the method of block decomposition and forming of SBP-SAT operators. In Section 3, we give performance evaluation of this problem on a single domain with different implementations. Namely, we confirm convergence results of SBP-SAT discretizaiton with different orders of accuracy. We test time and space resource consumption with different implementations. In Section 4, we cover the key ideas of the hybrid method introduced in the previous section, with extensive study on the performance of this novel method. We illustrate the promising aspects of this method and describe several existing issues of the current implementation. Solving these issues in the future implementation would be essential to further leveraging the benefits of the hybrid SBP-SAT scheme.

2 SBP-SAT introduction

2.1 One Dimensional SBP Operators

We discretize the domain $0 \leq x \leq 1$ with $N+1$ evenly spaced grid points $x_i = ih, i = 0, \ldots, N$ with spacing h = 1/N. We then project a function u onto the computational grid to be $u = [u_0, u_1, \ldots, u_N]^T$. u is often taken to be the interpolant of u at grid points. We define grid basis vector e_j to be a vector with value 1 at grid point j and 0 for the rest. We only need e_0 and e_N to form projections at boundaries. Note that in general we have $u_j = e_j^T u$.

We apply the class of high-order accurate SBP finite difference methods for first order derivatives which were introduced in Kreiss and Scherer (1974) and Kreiss and Scherer (1977) and Strand (1994) as mentioned above. For second order derivatives, we apply Mattsson and Nordström (2004), with variable coefficients treated in Mattsson (2012). The exact form of definitions are given below.

Definition 1 (First Derivative) We define matrix $\boldsymbol{D}_x$ to be an SBP approximation to $\partial u/\partial x$ if it can be decomposed as $\boldsymbol{H}\boldsymbol{D}_x = \boldsymbol{Q}$ with $\boldsymbol{H}$ being symmetric positive definite and $\boldsymbol{Q}$ satisfying $\boldsymbol{u}^T(\boldsymbol{Q}+\boldsymbol{Q}^T)\boldsymbol{v} = u_N v_N - u_0 v_0$.

Here, we only consider diagonal-norm SBP, i.e. finite difference operators where H is a diagonal matrix and D_x is the standard central finite difference matrix in the interior which transitions to one-sided at boundaries. The condition of Q defined above can be written as $\boldsymbol{Q}+\boldsymbol{Q}^T = \boldsymbol{e}_N\boldsymbol{e}_N^T - \boldsymbol{e}_0\boldsymbol{e}_0^T$.

The reason why the operator D_x is called SBP because it mimics the integration-by-part property

$$\int_0^1 u\frac{\partial v}{\partial x} + \int_0^1 \frac{\partial u}{\partial x}v = uv\Big|_0^1, \tag{3}$$

in a discrete form

$$\boldsymbol{u}^T\boldsymbol{H}\boldsymbol{D}_x\boldsymbol{v} + \boldsymbol{u}^T\boldsymbol{D}_x^T\boldsymbol{H}\boldsymbol{v} = \boldsymbol{u}^T\left(\boldsymbol{Q}+\boldsymbol{Q}^T\right)\boldsymbol{v} = u_N v_N - u_0 v_0. \tag{4}$$

Following the same pattern of the first derivative, we can define the second derivative.

Definition 2 (Second Derivative) We define matrix $\boldsymbol{D}_{xx}^{(c)}$ to be an SBP approximation to $\frac{\partial}{\partial x}\left(c\frac{\partial u}{\partial x}\right)$ if it can be decomposed as $\boldsymbol{H}\boldsymbol{D}_{xx}^{(c)} = -\boldsymbol{A}^{(c)} + c_N\boldsymbol{e}_N\boldsymbol{d}_N^T - c_0\boldsymbol{e}_0\boldsymbol{d}_0^T$ where $\boldsymbol{A}^{(c)}$ is symmetric positive definite and $\boldsymbol{d}_0^T\boldsymbol{u}$ and $\boldsymbol{d}_N^T\boldsymbol{u}$ are approximations of the first derivative of u at the boundaries.

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

$$\int_0^1 u\frac{\partial}{\partial x}\left(c\frac{\partial v}{\partial x}\right) + \int_0^1 \frac{\partial u}{\partial x}c\frac{\partial v}{\partial x} = uc\frac{\partial v}{\partial x}\Big|_0^1, \tag{5}$$

in a discrete form

$$\boldsymbol{u}^T\boldsymbol{H}\boldsymbol{D}_{xx}^{(c)}\boldsymbol{v} + \boldsymbol{u}^T\boldsymbol{A}^{(c)}\boldsymbol{v} = c_N u_N\boldsymbol{d}_N^T\boldsymbol{v} - c_0 u_0\boldsymbol{d}_0^T\boldsymbol{v}. \tag{6}$$

As noted above, we only consider diagonal-norm SBP finite difference operators here. In the interior, the operators use the minimal bandwidth central difference

stencil and transition to one-sided at boundaries in a manner that preserves the SBP property.

It has been known that for SBP operators defined above, if the interior operator has accuracy $2p$, then the interior stencil bandwidth is $2p+1$ and the boundary operator has accuracy p. If we use operators with interior accuracy $2p = 2$, 4, and 6, the expected global order of accuracy is the minimal of $2p$ and $p+2$ as evidenced by empirical study (Mattsson, Ham, et al. (2009) Virta and Mattsson (2014)) and proved for the Schrödinger equation (Nissen et al., 2013). We will verify the result in later sections.

2.2 Two Dimensional SBP Operators

Two-dimensional SBP operators can be developed by applying the one-dimensional SBP operators in a tensor product faction. Here we describe the operators for a rectangular block $\hat{B} \in [0,1] \times [0,1]$. We discretize this domain similar to 1d case in each direction resulting in an $(N+1) \times (N+1)$ grid of points where grid point (i,j) is at $(r_i, s_j) = (ih, jh)$ for $0 \leq i,j \leq N$ with $h = 1/N$; For simplicity, we only consider the case where we have the same numbers of grid points in each direction. A more complex scenario where we have different numbers of grid points in each direction can be formed similarly, but we are not going to discuss in detail here.

The two dimensional SBP operators can be obtained from one dimensional SBP operators by taking Kronecker products with them in orders that are determined by directions in two dimensional space. The detailed technique can be found in Kozdon, Erickson, et al. (2020) and won't be repeated here. We should note that tensor products are used here mainly for the purpose of simplicity in theoretical analysis. In computer memory, data are stored in a one dimensional array. Hence, the Kronecker products here mainly affects the order where we read data from a one dimensional array.

2.3 SAT Terms

SAT terms weakly enforce boundary conditions penalizing the grid point at the boundary towards the boundary data. It has the following simplified form:

$$\boldsymbol{b} = \alpha * (\mu \boldsymbol{B}\boldsymbol{u} - \boldsymbol{g}). \tag{7}$$

Here, $\boldsymbol{u}$ is the grid vector (the numerical approximation to the solution), $\boldsymbol{g}$ is boundary condition for a particular interfaces. $\boldsymbol{B}$ is an SBP operator that extracts boundary data from $\boldsymbol{u}$ and it would contain information about boundary layouts and associated conditions. μ is the block-diagonal matrix associated with $\boldsymbol{B}$ that needs to be compatible with boundary layouts. α is the penalty parameter in SAT term that is chosen under stability constraints from energy estimate. Finally, $\boldsymbol{b}$ is the assembled vector that weakly enforces a certain boundary condition in finite difference methods. More detailed examples of SAT terms in practice can be found in Erickson and Dunham (2014). Boundary conditions can be assembled by gradually adding $\boldsymbol{b}$ terms to the RHS of the equation in a simple additive way. Compared to the traditional method of using injection or strong enforcement of boundary/interface conditions that would destroy the SBP property defined in

equations 4 and 6 in section 2, using SAT terms preserves strict stability meaning that the semi-discrete problem has the same asymptotic time-growth as the continuous problem (Mattsson, 2003).

The combined SBP-SAT approach has been extensively used in computational science for solving problems from natural sciences where physical interfaces are ubiquitous. In geophysics particularly, it can be used to solve earthquake problems where continental and oceanic crustal blocks are separated by faults or in multi-phase fluids with discontinuous properties (Kozdon, Dunham, et al., 2012; Erickson and Day, 2016; Karlstrom and Dunham, 2016; Lotto and Dunham, 2015).

2.4 Implementation in Julia

In the Julia implementation, we use SparseArrays.jl to form sparse matrices. We use LinearAlgebra.jl for Kronecker products and other linear algebra operations. We use CUDAnative.jl and CuArrays.jl for computations on CUDA supported GPUs. For iterative solvers, we use IterativeSolvers.jl as well as our own matrix-free version of the CG algorithm. This project has been completely done in Julia except the meshing part where we use the external meshing software Trelis (`https://csimsoft.com/trelis`).

3 Performance Study on A Single Domain

3.1 Comparison Between Direct Solver and Iterative Solver

We first study the performance of the hybrid SBP-SAT method on a single domain. We chose a unit square, where we have Dirichlet boundary conditions on the left and the right, and Neumann boundary conditions on the top and bottom. To study the accuracy and convergence of the method, we apply the method of manufactured solutions (MMS), see Roache (1998) for example. In the MMS technique, an analytic solution is assumed from which we can derive compatible boundary and source data. In our test, we manufactured a solution to have the following form:

$$u(x,y) = sin(\pi x + \pi y), 0 \leq x \leq 1 \tag{8}$$

where x and y denote the x and y coordinates of a given point. From this fabricated solution, we can derive the conditions on the boundary and source function in interior, namely

$$\begin{cases} u_{xx} + u_{yy} + 2\pi^2 \sin(\pi x + \pi y) = 0, & 0 \leq x \leq 1 \quad (9a) \\ u = \sin(\pi y), & x = 0 \quad (9b) \\ u = -\sin(\pi y), & x = 1 \quad (9c) \\ -u_y = \cos(\pi x), & y = 0 \quad (9d) \\ u_y = -\cos(\pi x), & y = 1. \quad (9e) \end{cases}$$

Here u_y represents the first derivative of u with respect to y and u_{xx} represents the second derivative of u with respect to x. We don't have cross derivative terms

in our simplified problem. The minus sign from the quation 9d above comes in because the normal vector at $y = 0$ points downward. We define the $\text{error}_h = \sqrt{(u_h - u)^T H (u_h - u)}$. Here u is the exact solution in equation 8 evaluated on the numerical grid, and u_h stands for numerical results from solving the system defined by and boundary conditions and source functions obtained from fabricated results u with governing equations in 9. u_h and u are stacked to be one-dimensional vectors. H is SBP operator that incorporates grid space information and is the Kronecker product of H_x and H_y defined in the 2D SBP operators. This definition of error_h is the discrete-L2 error and is used for convergence tests.

We begin convergence tests using a direct solver on CPU. For different p values, we obtain the following accuracy results for convergence tests in table 1. We obtained expected convergence results for the 2nd order and 4th order SBP operators. For 6th order SBP operators, the convergence rates are close to 5.5. The reason why in higher order operators we don't observe convergence rate as the order of SBP operators is because the order of the accuracy is lower on boundaries as described in 2. Also we are reaching machine precision with $p = 6$, so the rate of convergence is also affected by this factor.

	2nd Order		4th Order		6th Order	
N	error_N	rate	error_N	rate	error_N	rate
2^4	1.735×10^{-3}		4.227×10^{-5}		1.139×10^{-5}	
2^5	4.319×10^{-4}	2.0013	2.117×10^{-6}	4.320	2.605×10^{-7}	5.451
2^6	1.079×10^{-4}	2.0003	1.095×10^{-7}	4.273	5.847×10^{-9}	5.477
2^7	2.696×10^{-5}	2.00007	5.956×10^{-9}	4.200	1.301×10^{-10}	5.489
2^8	6.740×10^{-6}	2.000017	3.401×10^{-10}	4.130	2.896×10^{-12}	5.489

Table 1: Error and convergence rates using the method of manufactured solutions.

Convergence results above have verified the correctness of our implementations. We can also verify this with numerical results from the iterative solvers with similar outcomes.

Although our ultimate goal is to solve this problem on a very large system, once convergence is verified, our next question is how can we solve this problem more efficiently while maintaining correct results. We fix $p = 2$ to reduce the number of variables in our study. Our linear system has a positive semi-definite (PD) left-hand-side (LHS). It is easy to verify that the CG method out performs other iterative solvers that are designed to handle non-PSD cases such as MINRES (for indefinite matrices) or GMRES (for non-symmetric matrices when good preconditioning is available). We now compare the performance of a direct solver with the CG method on both CPU and GPU. The accuracy results are shown in table 2 to demonstrate all three methods succeed in producing correct results. We should note that by default GPU works with Float32 which normally has significantly higher peak FLOPS than Float64. For convergence and accuracy comparisons however, we chose Float64 on GPU to compare with Float64 on CPU32. Float32 on GPU still yields rather high accuracy (up to 2^{-9}) which can be sufficient enough (depending on our accuracy requirements) while obtaining optimal performance. We evaluate the performance here according to how long it takes to solve the lin-

Grid Amounts	Direct Solver	GPU Iterative	CPU Iterative
N	$\text{Log}_2(\text{error}_N)$	$\text{Log}_2(\text{error}_N)$	$\text{Log}_2(\text{error}_N)$
2^3	-7.140701	-7.140701	-7.140701
2^4	-9.170836	-9.170835	-9.170836
2^5	-11.17709	-11.177089	-11.177089
2^6	-13.178418	-13.178418	-13.178461
2^7	-15.178715	-15.178781	-15.178781

Table 2: Log Errors (base 2) Comparison of Direct Solver and Iterative Solver

ear system and how much memory is allocated. In order to achieve reproducible results, we use the zero vector as the initial guess for the CG method. In practice, using a random initialization normally gives 10x speed-up on our problem. We use the BenchmarkTools.jl package for performance evaluation.

N	Direct Solver	GPU Iterative	CPU Iterative	Direct Solver After LU
2^3	91.387 μs	143.526 μs	1.112 μs	3.082 μs
2^4	313.194 μs	158.186 μs	4.249 μs	12.953 μs
2^5	1.180 ms	165.876 μs	10.934 μs	58.580 μs
2^6	5.799 ms	599.566 μs	37.629 μs	271.836 μs
2^7	32.527 ms	12.113 ms	144.675 μs	1.308 ms

Table 3: Time Comparison of Direct Solver and Iterative Solver

N	Direct Solver	GPU Iterative	CPU Iterative	Direct Solver After LU
2^3	68.99 KiB	5.64 KiB	2.97 KiB	1.47 KiB
2^4	250.24 KiB	5.67 KiB	8.13 KiB	4.91 KiB
2^5	988.21 KiB	5.75 KiB	26.91 KiB	17.41 KiB
2^6	4.03 MiB	5.75 KiB	100.27 KiB	66.31 KiB
2^7	22.27 MiB	2.93 MiB	651.58 KiB	260.31 KiB

Table 4: Memory Comparison of Direct Solver and Iterative Solver

After we verified the correctness, we compare the performance of the direct and iterative methods in terms of time and memory. For iterative methods, we use built-in CG method in IterativeSolvers package. For hyperparameters in CG, we use default ones. The maximum number of iterations is chosen to be the size of the linear system and the tolerance is set to be the square root of machine epsilon for a given floating point type. The results are given in table 3 and 4 respectively. Within the context of a time stepping method, the cost of the LU factorization is overhead cost; once it's obtained, it can be reused in direct solver. Therefore, to compare the actual performance of the iterative solver with direct solver, excluding the cost of LU factorization is an important consideration when working with a time-dependent problem. However we should note that the size of the system that we can solve is limited by the memory cost from LU decomposition. As we can see from the table 4 and 3, for our system, even with factorization cost removed,

the iterative solver on CPU still out-performs the direct solver both in time and memory for the same problem. This nice speedup from using the CG iterative solver is most likely coming from the fact that our matrix is PD. Another nice property of our system is the narrow-band sparsity, which requires significantly less computational cost with iterative solvers compared to having factorization that cannot preserve sparsity. Both iterative solvers on CPU and GPU would use parallel framework such as BLAS and CUDA to accelerate, which is less practical in factorization. All these three contribute to the fact that even though factorization results can be reused, solving this system with direct method is still much more expensive than using iterative solvers.

3.2 Further Speedup of Iterative Solvers

One nice feature of using an iterative solver is the access to matrix-free methods where instead of forming matrix A, we compute the product Ax. We do this by writing a function $f(x)$ that takes vector x as input and directly modifies the entries of x to achieve the same results of Ax. A naive example would be when A is identity matrix, we can use $f(x) = x$ that export x itself as output which has the same effect as multiplying x by identity matrix A in Ax. A matrix-free implementation is extremely handy when the matrix A is sparse and is close to diagonal or tridiagonal with very low bandwidth such as we have in the hybridized SBP-SAT method. Using matrix free methods reduces the memory allocation from forming matrix A. And in-situ operations on x are faster in some cases by reducing the cost of forming intermediate results and storing them in matrix multiplications. One thing we need to be careful about is that Julia's hidden pointer mechanism makes it prone to data contamination when we tried to use some of Julia's notations for fast I/O. The solution is similar to writing Julia in C++ where you need to determine the data containers needed in advance and allocate them in memory, then reuse these containers in matrix-free functions. Our implementation showed that matrix-free functions achieve speedup of several times to ten times compared to sparse-matrix counterparts with zero garbage collector (GC) times. The zero overhead is extremely important because for large matrix operations, GC would determine the maximum time to finish a task, which would cause the volatility in run-time behavior depending on system load. For ideal parallelization, we would expect less volatility in order to have nice static load balancing in designing parallel scheme with optimal performance. Our matrix-free method can be easily parallelized using built-in multi-threading macros in Julia.

To demonstrate the speed-up of multi-threading in Julia, we compared two versions of the same matrix-free functions, one with multi-threading and one in serial. The only difference is that in the multi-threaded matrix-free function, we add the @threads macro before the for loops. We tested speedup from multi-threading with respect to different system sizes. In our local environment, we set the number of threads in Julia to be 4. The results are shown in the table 5. Multi-threading has fixed overhead which makes it more expensive when our system is small, but as the size of the problem increases, this overhead is negligible and we can see significant speed-up from even naive implementation using the built-in macro. To further explore the speedup we can achieve by throwing more threads, we tested the case where $N = 10000$ on the Talapas server (https://datascience.

uoregon.edu/talapas-supercomputer) . Using 32 threads, we achieve 10x speedup. As the system size increases, We are expected to see more speedup from the multi-threading technique.

	Serial		Multithreading	
N	Time	Memory	Time	Memory
100	56.014 μs	79.83 KiB	119.677 μs	87.33 KiB
1000	2.965 ms	7.64 MiB	2.173 ms	7.65 MiB
10000	809.077 ms	763.09 MiB	423.203 ms	763.10 MiB

Table 5: Performance of Multi-threading in Matrix-free Operators

The speedup from matrix-free operators themselves associated with feasible multi-threading in matrix-free methods make it more appealing when using iterative solvers compared to sparse-matrix formulations. However, one big challenge is to optimize the memory allocation and multi-threading in a compound functions that calls external functions. Our matrix-free CG method out-performs existing sparse-matrix CG implementations, but the speed-up is not ideal to what we expected from the speedup from consisting functions within our CG. Further optimization would be needed to further leveraging the power of matrix-free method and parallel processing including multi-threading.

4 Hybridized SBP Scheme

One of the main goals of this work is to explore peformance gains using the newly proposed hybridized SBP-SAT scheme from Kozdon, Erickson, et al. (2020). In the finite element literature, a hybrid method is the method where one unknown is a function on the interior of the elements and the unknown is function on the trace of the elements (Ciarlet, 2002, page 421). For SBP methods particularly, we write the method in terms of local problems and associated global problem. In the local problems, for each block $B \in$ a grid of blocks $\mathcal{B}$, the trace of the solution (i.e., the boundary and interface data) is assumed and each set of equations (2) is solved locally over B. In the global problem the solution traces for each $B \in \mathcal{B}$ are coupled. This technique will result in a linear system of the form

$$\begin{bmatrix} \bar{\boldsymbol{M}} & \bar{\boldsymbol{F}} \\ \bar{\boldsymbol{F}}^T & \bar{\boldsymbol{D}} \end{bmatrix} \begin{bmatrix} \bar{\boldsymbol{u}} \\ \bar{\boldsymbol{\lambda}} \end{bmatrix} = \begin{bmatrix} \bar{\boldsymbol{g}} \\ \bar{\boldsymbol{g}}_\delta \end{bmatrix}. \tag{10}$$

Here $\bar{\boldsymbol{u}}$ is the approximate solution to (2) at all the grid points and $\bar{\boldsymbol{\lambda}}$ are the trace variables along internal interfaces; trace variables that are associated with boundary conditions can be eliminated. The matrix $\bar{\boldsymbol{M}}$ is block diagonal consisting symmetric positive definite blocks for each $B \in \mathcal{B}$, $\bar{\boldsymbol{D}}$ is diagonal, and the matrix $\bar{\boldsymbol{F}}$ is sparse and incorporates the coupling conditions. The right-hand side vector $\bar{\boldsymbol{g}}$ incorporates both boundary data (g_D, g_N) and source terms whereas $\bar{\boldsymbol{g}}_\delta$ incorporates the interface data δ.

Using the Schur complement we can transform (10) to (11a) and (11b). The problem size in (10) is significantly reduced since the number of trace variables is

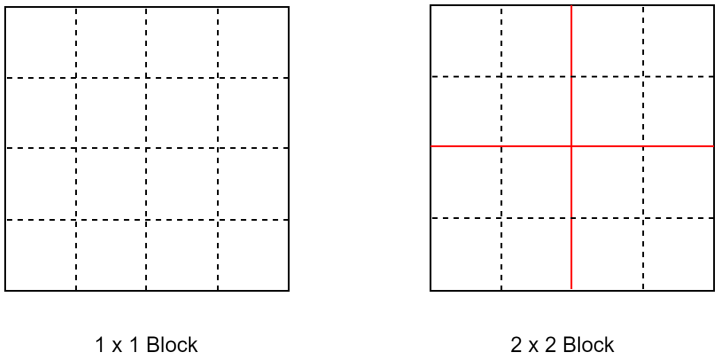

Fig. 2: Sample 2D meshes with different block numbers

significantly smaller than the number of solution variables. Because $\bar{\boldsymbol{M}}$ is block diagonal here, the inverse of a block diagonal matrix can be obtained in a decoupled manner for each $B \in \mathcal{B}$. Thus there is a trade-off between the number of blocks and the size of system (11a), since for a fixed resolution increasing the number of blocks means that $\bar{\boldsymbol{M}}$ will be more efficiently factored but the size of (11a) will increase through the introduction of additional trace variables.

$$\left(\bar{\boldsymbol{D}} - \bar{\boldsymbol{F}}^T \bar{\boldsymbol{M}}^{-1} \bar{\boldsymbol{F}}\right) \bar{\boldsymbol{\lambda}} = \bar{\boldsymbol{g}}_\delta - \bar{\boldsymbol{F}}^T \bar{\boldsymbol{M}}^{-1} \bar{\boldsymbol{g}}, \tag{11a}$$

$$\left(\bar{\boldsymbol{M}} - \bar{\boldsymbol{F}} \bar{\boldsymbol{D}}^{-1} \bar{\boldsymbol{F}}^T\right) \bar{\boldsymbol{u}} = \bar{\boldsymbol{g}} - \bar{\boldsymbol{F}} \bar{\boldsymbol{D}}^{-1} \bar{\boldsymbol{g}}_\delta. \tag{11b}$$

The big picture of the hybridized method is described above, more detailed formulation of how local problems and global problems are formed can be found in Kozdon, Erickson, et al. (2020). For the problem that we are studying, we want to see the trade-off between the number of the blocks and the number of trace variables, and how this trade-off would affect the performance of solving Poisson's equation 9 on a given domain. For evaluation, we used and modified the code in `https://github.com/bfam/HybridSBP` for our test. We generate different square meshes that have unit length with decreasing grid spacing using built-in mesh refinement in the Trelis meshing software. For a unit square domain, we can choose different numbers of blocks with different local meshing levels within each block to achieve the same number of total grid points in each direction. To demonstrate this, we use two different meshes with the same number of total grid points in each direction in figure 2. The first mesh has only one block with no trace variables, and each block has 2^2+1 grid points in each direction. The second mesh has four blocks with internal boundaries marked in red, and each block has 2^1+1 grid points. Trace variables along these internal boundaries can be obtained from solving a global problem using Schur complement as described previously.

We use mesh refinement to generate unit mesh with the number of blocks in each direction ranging from 1 to 2^{10} in Trelis. Within each block, we start with 2^4 and gradually start mesh refinement by increasing the number of grid points in each direction and creating the respective local operators. Note that this is

Number of Grid Points	Block Size Of Given Mesh				
N	1 x 1 (LU Factorization)	2 x 2	4 x 4	8 x 8	16 x 16
2^4	✓				
2^5	✓	✓			
2^6	✓	✓	✓		
2^7	✓	✓	✓	✓	
2^8	✓	✓	✓	✓	
2^9	✓	✓	✓	✓	
2^{10}	✓	✓	✓	✓	
2^{11}	✓	LoadError	LoadError	✓	✓
2^{12}	MallocError	×	×	?	?

Table 6: The largest system that can be solved with different number of blocks

different than in mesh refinement software where increasing the number of blocks also increases the number of interfaces. Mesh refining in Julia only applies to local operators, with number of interfaces fixed. The number of trace variables would increase because each interface now have more trace variables as a result of local block mesh refinement. The first question we are trying to answer is whether the hybrid-method with multiple blocks can help us solve a problem with more grid points in each direction without using iterative solvers under the constraint of available memory. We choose 6th order operators for comparison. We requested 128 GB memory on the Talapas server and explored the largest problems we can solve with respect to different block sizes using the hybrid method with a direct solvers. We also added results from a single block as benchmark. The results are given in table 6

In this table, N+1 refers to the total number of grid points in each direction. The MallocError in the table refers to the memory allocation errors that we encountered in the LU decomposition. The LoadError in the table refers to the error of loading the Cholesky factorization which is used in the hybridized method. They are denoted differently but the reasons are all associated with the limitation of memory for factroizations. We are limited by the size of the matrices that we can factorize. Note that our non-hybrid implementation on single block previously used LU factorization instead of Cholesky factorization. The results between a single block with multi-blocks can be inconsistent. But we can see as we increase the number of the blocks, we are able to solve this problem on a larger system. The reason that we couldn't finish working on the server is limited by the job length that we submitted. The calculation for 2^{12} block was terminated for both cases. No factorization error was reported after successfully obtaining results for 2^{11}. This means that given enough time, we are expected to see the system being solved with 2^{12} grid points along each side on an 8×8 block. This is further confirmed that even though both the 8×8 grid of blocks and 16×16 grid of blocks can be used to solve a system with $2^{11} + 1$ total grid points in each direction, the memory allocations on the latter one is much less than the previous one because the most computationally expensive part is factorization, and in hybrid scheme, this is associated with local block sizes. The number 8 in 8×8 here refers the number of the grid points is $2^8 + 1$. If there is no mesh refinement within each grid, the associated linear system has the size of 2^{16} by 2^{16}.

N	Grid Size Of Given Mesh								
	4 x 4			8 x 8			16 x 16		
	N_p^{Vol}	N_p^{Tr}	N_p^{Vol}/N_p^{Tr}	N_p^{Vol}	N_p^{Tr}	N_p^{Vol}/N_p^{Tr}	N_p^{Vol}	N_p^{Tr}	N_p^{Vol}/N_p^{Tr}
2^6	4624	408	11.33	5184	1008	5.14	6400	2400	2.67
2^7	17424	792	22	18496	1904	9.71	20736	4320	4.8
2^8	67600	1560	43.33	69696	3696	18.86	73984	8160	9.07
2^9	266256	3096	86	270400	7280	37.14	278784	15840	17.6
2^{10}	1056784	6168	171.33	1065024	14448	73.71	1081600	31200	34.67
2^{11}	4210704	12312	342	4227136	28784	146.86	4260096	61920	68.8

Table 7: Number of volume points and trace variables with different number of blocks

The reason why we are seeing significantly long run-time from the code is because in the original implementation of the hybrid method, there is no parallel mechanism involved. Ideally, since each block is decoupled after solving the global problem, the factorization for each block can be done independently. Also in the hybrid method, many interior blocks have the same boundary conditions, which results in identical SBP operators on the LHS for these blocks. Therefore there is no need to do factorization for each matrix on the LHS. We only need to compute a factorization for each distinctive local matrix and reuse the result when we encounter an identical one. This is also not implemented in the code. The work of performance improvement is crucial to not just being able to solve a larger problem, but also solving it faster. We will continue exploring the optimization of the hybrid method. Nevertheless, with larger block-sizes, we can still solve the problem faster with more blocks. For direct solve after obtaining factorization results, we can also use multi-threading for acceleration. We set number of threads to be a fixed value 6, and we evaluate performance on meshes with different block numbers. To avoid the impact of jobs being assigned to different node with different capacity, we tested performance on a local computer with i-9500f CPU and fixed 16 Gb memory. The result is given in the table 8. For comparison we also list the number of volume points and trace variables in table 7. They are defined as:

$$N_p^{vol} = (N_l + 1)^2 N_b$$
$$N_p^{tr} = (N_l + 1) N_I$$

Here N_b represents the number of blocks, N_l refers to the number of grid points in each direction in the local blocks. N_I represents the number of interfaces. The ratio N_p^{tr}/N_p^{vol} would increase for the same N as we increase the number of blocks, which can be confirmed with data in table 7.

From table 8 , we can see as we increase the number of blocks, for the same N, more numbers of blocks mean more local systems to solve, but the total time to solve all these local system is less. The computational complexity of direct solvers after factorization is $O(N^2)$. (Here N means the size of the system, not number of grid points in each direction). For each N, doubling the number of blocks in each direction would reduce the size of the local system to 1/4. The cost of solving each local problem is 1/16 of the previous one. But we also have 4 times more local systems. The total cost should be reduced by 1/4. We can see the time to do the direct solve after LU decomposition is faster, but the speedup is not 4x.

Number of Grid Points	Grid Size Of Given Mesh		
N	4 x 4	8 x 8	16 x 16
2^6	0.00410 s		
2^7	0.00270 s	0.00510 s	
2^8	0.01370 s	0.00920 s	0.00840 s
2^9	0.05480 s	0.02920 s	0.021300 s
2^{10}	0.29040 s	0.19320 s	0.11650 s
2^{11}	LoadError	1.12210 s	0.89510 s

Table 8: Time for direct solvers after factorization with different number of blocks

Number of Grid Points	Grid Size Of Given Mesh		
N	4 x 4	8 x 8	16 x 16
2^6	1251.687673025973		
2^7	0.2722459250042488	8076.298905171896	
2^8	0.006891702238438522	0.006643733883857528	11821.804240653208
2^9	0.0001688738908282545	0.0001429235518197867	0.0017873144253085962
2^{10}	3.932605990678488e-6	2.7931022138805148e-6	3.6928926337184484e-6
2^{11}	LoadError	7.434960489253316e-8	8.87865211871948e-8

Table 9: Errors for direct solvers with different number of blocks

The reason could be from implementation side, or it is coming from the fact that we are solving a sparse system instead of a dense system. Further experiments are needed to answer this question.

We need to further verify that using different number of blocks in hybrid method can give results with errors at the same level if not identical. We listed results in table 9. Errors for the coarsest mesh on different blocks are significantly larger than the rest, this is because of the test function we used in Kozdon, Erickson, et al., 2020. As we continue further mesh refinement in each local block, we would obtain sigfinicantly reduced errors that are also converging.

The results are shown in table 10. The cost for forming the global problem and solving for trace variables, although not optimized with parallel scheme, also favors using more blocks for a given N that is sufficiently large. But for smaller N, further increasing the number blocks increases the number of trace variables in the global problem, hence solving the global problem becomes the bottle-neck for the whole problem. This can be seen when N=2^8 and 2^9, using 16×16 blocks results in worse performance compared to using 8×8 block. With optimized parallel implementation leveraging the advantage of having more decoupled local systems, we would expect for a very large system size that the hybrid SBP method would outperform the non-hybrid SBP method on a single domain. The fact that each local system is decoupled makes this problem embarrassingly parallel, which is extremely suitable for architecture that is better for computationally intensive tasks rather than I/O intensive tasks.

5 Conclusions

We tested the SBP-SAT method in solving Poisson's equation on a single domain. Our results show that the Conjugate Gradient method on CPU outperforms Con-

Number of Grid Points	Grid Size Of Given Mesh		
N	4 x 4	8 x 8	16 x 16
2^6	0.202 s		
2^7	1.460 s	0.005 s	
2^8	3.961 s	2.374 s	4.650 s
2^9	34.330 s	19.740 s	29.408 s
2^{10}	307.085 s	183.804 s	167.213 s
2^{11}	LoadError	1522.338 s	1170.490 s

Table 10: Time for solving global problems with different number of blocks

jugate Gradient methods on GPU and direct solvers on CPU even excluding factorization costs. We weren't able to test direct methods on GPU in our previous test due to the lack of properly implemented Julia interface to CUDA libraries. GPU iterative solvers have high start-up costs which make them not suitable for solving Poisson equations on a small system. We were limited by the incompatibility of Julia packages in GPU computing and linear algebra and we failed to test further on a large system. But computations on GPU are more scalable for linear algebra operations and also more ideal for computationally intensive jobs that are bottle-necked by I/O through-put between GPU and CPU. We are able to test more thoroughly with recent updates in Julia and related packages that fix the existing issues that limit our previous implementation.

We also tested the newly proposed hybrid SBP scheme. Our result shows that hybrid SBP method can work on a larger system compared to traditional non-hybridized SBP method under the constraint of system memory. Our empirical results also show that the hybrid method achieves better performance on a system with more blocks. For certain values of N, using more grid points at the start improves performance, but further increasing the number of blocks reduces the performance. To explore the trade-off relationship between the number of trace variables and the local problem size quantitatively, we need to test our problem on more grids with different numbers of blocks.

We bench-marked existing implementations to identify several issues that limit the size of the system that we are able to solve. We tested different methods to improve the current implementation. Results from our study form the foundation of utilizing the new finite difference method whose hybrid structure is ideally suited for GPU parallelization to achieve performance gains. This is much needed for simulations on a large problem which would give rise to a extremely large linear system that can't be solved with existing method and implementation, in order to understand the behaviors of earthquakes in real scenario.

Acknowledgement

This work benefited from access to the University of Oregon high performance computer, Talapas.

References

Besard, T., C. Foket, and B. De Sutter (2019). "Effective extensible programming: unleashing julia on gpus". In: *IEEE Transactions on Parallel and Distributed Systems* 30.4, pp. 827–841.

Carpenter, M. H., D. Gottlieb, and S. Abarbanel (1994). "Time-stable boundary conditions for finite-difference schemes solving hyperbolic systems: methodology and application to high-order compact schemes". In: *Journal of Computational Physics* 111.2, pp. 220–236. DOI: 10.1006/jcph.1994.1057.

Carpenter, M. H., J. Nordström, and D. Gottlieb (1999). "A stable and conservative interface treatment of arbitrary spatial accuracy". In: *Journal of Computational Physics* 148.2, pp. 341–365. DOI: 10.1006/jcph.1998.6114.

Ciarlet, P. G. (2002). *The Finite Element Method for Elliptic Problems*. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898719208.

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

Erickson, B. A. and E. M. Dunham (2014). "An efficient numerical method for earthquake cycles in heterogeneous media: alternating subbasin and surface-rupturing events on faults crossing a sedimentary basin". In: *Journal of Geophysical Research: Solid Earth* 119.4, pp. 3290–3316. DOI: 10.1002/2013JB010614.

Karlstrom, L. and E. M. Dunham (2016). "Excitation and resonance of acoustic-gravity waves in a column of stratified, bubbly magma". In: *Journal of Fluid Mechanics* 797, pp. 431–470. DOI: 10.1017/jfm.2016.257.

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

Kozdon, J. E., B. A. Erickson, and L. C. Wilcox (2020). "Hybridized summation-by-parts finite difference methods". In: *ArXiv* abs/2002.00116.

Kreiss, H. and G. Scherer (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*. Madison, WI, pp. 195–212. DOI: 10.1016/b978-0-12-208350-1.50012-1.

— (1977). *On the existence of energy estimates for difference approximations for hyperbolic systems*. Tech. rep. Department of Scientific Computing, Uppsala University.

Lotto, G. C. and E. M. Dunham (2015). "High-order finite difference modeling of tsunami generation in a compressible ocean from offshore earthquakes". In: *Computational Geosciences* 19.2, pp. 327–340. DOI: 10.1007/s10596-015-9472-0.

Mattsson, K. (2012). "Summation by parts operators for finite difference approximations of second-derivatives with variable coefficients". In: *Journal of Scientific Computing* 51.3, pp. 650–682. DOI: 10.1007/s10915-011-9525-z.

Mattsson, K., F. Ham, and G. Iaccarino (2009). "Stable boundary treatment for the wave equation on second-order form". In: *Journal of Scientific Computing* 41.3, pp. 366–383. DOI: 10.1007/s10915-009-9305-1.

Mattsson, K. and J. Nordström (2004). "Summation by parts operators for finite difference approximations of second derivatives". In: *Journal of Computational Physics* 199.2, pp. 503–540. DOI: 10.1016/j.jcp.2004.03.001.

Mattsson, K. (Feb. 2003). "Boundary procedures for summation-by-parts operators". In: *Journal of Scientific Computing* 18.1, pp. 133–153.

Nissen, A., G. Kreiss, and M. Gerritsen (2013). "High order stable finite difference methods for the Schrödinger equation". In: *Journal of Scientific Computing* 55.1, pp. 173–199. DOI: 10.1007/s10915-012-9628-1.

Roache, P. (1998). *Verification and Validation in Computational Science and Engineering*. 1st ed. Albuquerque, NM: Hermosa Publishers.

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. Second. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898718003. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898718003.

Strand, B. (1994). "Summation by parts for finite difference approximations for d/dx". In: *Journal of Computational Physics* 110.1, pp. 47–67. DOI: 10.1006/jcph.1994.1005.

Virta, K. and K. Mattsson (2014). "Acoustic wave propagation in complicated geometries and heterogeneous media". In: *Journal of Scientific Computing* 61.1, pp. 90–118. DOI: 10.1007/s10915-014-9817-1.