A high-performance characterization of hybridized PDEs

JOSEPH McLAUGHLIN
University of Oregon. Eugene, USA

Hybridization allows numerical partial differential equations (PDEs) to be broken up into smaller problems *i.e.*, elements, that can be solved independently and with less memory than with a single equivalent system. Methods such as matrix-free assembly can also reduce memory usage, but these methods are less flexible for anisotropic problems. In this work we provide a theoretical analysis of a hybridized, 2-D Poisson problem assembled with an summation-by-parts operator with simultaneous approximation terms (SBP-SAT). We also include an empirical evaluation of CPU implementation of the problem utilizing the PETSc library for linear algebra operations and OpenMP for parallelism. We demonstrate how the parameterization of the problem, including the number of elements and the size of each element, can effect the time-to-solution by an order of magnitude.

1 INTRODUCTION

Numerical solutions to linear elliptic partial differential equations (PDEs) are widely sought across mathematics, engineering, and computer science and are commonly used to model physical systems in steady-state or time-independent forms. Performant and flexible approaches of encoding increasingly large systems are critical as scale of computing resources continue to grow. Elliptical PDE problems involve setting up and solving a numerical linear system of equations which tend to be dominated by memory-bound operations. Modern applications that utilize PDEs in scientific simulations are increasingly complex, and need to consider everything from the machine to the abstract mathematics to effectively utilize available resources.

Hybridization *i.e.*, Guyan reduction [13] or static condensation [25], is one approach to reducing the computational complexity of computational systems. Hybridization is more commonly utilized using finite elements [10, 11, 21], but recent work from Kozdon et al. [18] has brought it into finite differences as well. This method reduces the complexity of the computational system, assembled as a system of coupled sub-domains that can each be solved independently alongside an additional global system. In a shared memory context this independence allows problems to be computed that would otherwise utilize too much memory.

Conventionally, this has lent hybridization the reputation of being mostly a tool for coupling otherwise unwieldy problems together, and less a consideration of performance. Either, a problem is too large to fit into memory and is Hybridized into smaller problems, or multiple problems that would otherwise be difficult to represent in a single domain is coupled via Hybridization.

In contrast to the conventional wisdom, recent work from Dean et al. [8] intentionally utilizes hybridization to demonstrate strong and weak scaling on a massively parallel problem assembled as a discontinuous Galerkin method. Another paper from Badrkhani et al. [2] utilizes hybridization to couple large matrix-free sub-domains together for competitive performance results.

The flexibility of hybridization makes it a compelling choice for a computing large parallel problems. Each sub-domain in a Hybridized problem is solved independently, though several of the *building blocks* needed to assemble a local problem can be shared between problems further reducing the computational complexity. However, increasing the number of local problem has the effect of increasing the size of the global problem. It follows that there should exist a trade off between the number of local problems and cost of computing the global system. Numerical analysis commonly deals with the numerical accuracy of a problem, but rarely considers time-to-solution a property of problem. Yet, there exists fundamental relationship between the scale and orientation of a computation and performance.

In this work we explore that trade off in a shared-memory CPU-based problem. We assemble a problem in a hybridized summation-by-parts with simultaneous-appropriation-terms (SBP-SAT) finite difference scheme, solving the 2-D Poisson equation, coupling multiple sub-domains with penalty terms analogous to Dirichlet boundary conditions *i.e.*, numerical fluxes as given in finite elements, proportionally varying the volume points within each sub-domain as to fix the number of total volume points. We implement this in PETSc [3] and view performance as a function of the number of volume points in each element *i.e.*, the element size, yielding a strong and weak scaling analysis. Throughout this work the terms sub-domain and element are used interchangeably to refer to the coupled problems within a hybridized problem.

2 BACKGROUND & MOTIVATION

In this section, We motivate this work with a brief review of related approaches to solving linear elliptic PDEs on parallel architectures. Additionally, we provide a description of the 2-D Poisson Equation that we study in this work. Finally, we provide a background on the hybridization, as well as with the SBP-SAT method that we use in the assembly of our problem.

2.1 Problem description

Through the use of the hybridized SBP-SAT method, we investigate the shared memory, strong-scaling characteristics of the 2D Poisson equation assembled with the SBP-SAT method [18]. Whereas PDE performance is largely considered in a weak-scaling context, we study the strong-scaling characteristics by fixing the number of volume points, leveraging the hybrid formulation of the SBP-SAT method, to solve a varied size and quantity of independent problems.

We assemble a 2D Poisson equation

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)u(x,y) = f(x,y) \tag{1a}$$

defined over the domain

$$0 \leq x \leq 1, \tag{1b}$$

$$0 \leq y \leq 1, \tag{1c}$$

given the boundary conditions

$$u(0,y) = \sin(y), \tag{1d}$$

$$u(1,y) = \sin(\pi + y), \tag{1e}$$

$$\frac{\partial u(x,0)}{\partial y} = -\pi\cos(\pi x), \tag{1f}$$

$$\frac{\partial u(x,1)}{\partial y} = -\pi\cos(\pi x) \tag{1g}$$

and source function

$$f(x,y) = -2\pi^2 u(x,y). \tag{1h}$$

This sets up a problem utilizing the manufactured solution

$$u(x,y) = sin(\pi x + \pi y). \tag{2}$$

2.2 Discretization

Discrete numerical formulations of the 2-D Poisson equation typically take the form

$$\mathbf{A}\mathbf{u} = \mathbf{b}, \tag{3}$$

where $\mathbf{A}$ is a discrete analogue of the Laplacian operator in Eq. 1a, $\mathbf{b}$ is a vector encoding the source function, f, and $\mathbf{u}$ is an approximate solution, order-accurate to the exact solution, u. Order of accuracy is typically determined by $\mathbf{A}$ and varies on the choice of method. In this problem we assemble $\mathbf{A}$ with the SBP-SAT method given in detail by Chen [7]. The SBP-SAT method has practical characteristics of high order accuracy and a proof of energy conservation. These qualities allow the SAT terms to act analogous to fluxes in discontinuous Galerkin methods (DGM), and as such allowed Kozdon et al. [18] to integrate the SBP-SAT method into a hybridized scheme.

2.3 Hybridized scheme

The hybridized SBP-SAT formulation discretizes this problem into a mesh of elements. This enables the solution to each element to be computed independently after solving a global, trace system. This is written in the form

$$\begin{bmatrix} \mathbf{M} & \mathbf{F} \\ \mathbf{F}^\intercal & \mathbf{D} \end{bmatrix}\begin{pmatrix} \mathbf{u} \\ \lambda \end{pmatrix} = \begin{pmatrix} \bar{\mathbf{g}} \\ \bar{\mathbf{g}}_\delta \end{pmatrix}. \tag{4}$$

Here, $\mathbf{u}$ remains a vector of the approximate solution to the grid points of the problem. λ is a vector of the trace variables along the internal interfaces of the system. $\mathbf{M}$ resembles $\mathbf{A}$, but is *block diagonal*, with each block encoding an element in the system. This is a key quality of hybridization, allowing the system to computed in a series of operations only dependent the alignment of each block. $\mathbf{F}$ is a sparse matrix containing the internal interface boundary coefficients, and $\mathbf{D}$ is sparse diagonal matrix. $\bar{\mathbf{g}}$ and $\bar{\mathbf{g}}_\delta$ contain the source data for the external boundaries and internal interfaces. We derive this hybridized system from Kozdon et al. [18], who provide a detailed description of the hybridized system.

From the Schur complement we have

$$(\mathbf{D} - \mathbf{F}^\intercal\mathbf{M}^{-1}\mathbf{F})\lambda = \bar{\mathbf{g}}_\delta - \mathbf{F}^\intercal\mathbf{M}^{-1}\bar{\mathbf{g}}. \tag{5}$$

This is solved in two parts: the global problem,

$$\lambda_\mathbf{A} = \mathbf{D} - \mathbf{F}^\intercal\mathbf{M}^{-1}\mathbf{F}, \tag{6a}$$

$$\lambda_\mathbf{b} = \bar{\mathbf{g}}_\delta - \mathbf{F}^\intercal\mathbf{M}^{-1}\bar{\mathbf{g}}, \tag{6b}$$

$$\lambda = \lambda_\mathbf{A}^{-1}\lambda_\mathbf{b}, \tag{6c}$$

and the local problem,

$$\mathbf{u} = \mathbf{M}^{-1}(\bar{\mathbf{g}} - \mathbf{F}\lambda). \tag{7}$$

This permits us **(a)** to compute much larger problems with less memory, especially if the problem is largely homogeneous, and **(b)** to reduce the computational complexity of solving the system by instead solving several smaller systems. In both cases this allows us express the Eq. (6a–6c, 7) as a concatenation of smaller problems.

To form the smaller problems we decompose the sub-matrices $\mathbf{M}$, $\mathbf{F}$, $\mathbf{F}^\intercal$, and $\mathbf{D}$. For example, as $\mathbf{M}$ is block-diagonal, we store each non-zero component in $\mathbf{M}$ as a matrix. For homogeneous problems several of

the non-zero components in $\mathbf{M}$ are identical, requiring only one copy for each unique component for a given shared-memory context. A similar structure exists for $\mathbf{F}$ and $\mathbf{F}^\intercal$, though the homogeneity of these matrices depends on the mesh structure, not the domain structure.

> For this work we consider a specialization of the problem where both the mesh and each element are square. Here, the total number of elements, ℓ^2, is the square of the number of elements on any face of the *global* domain, ℓ. Additionally, the number of volume points in each element, n^2, is the square of the number of volume points along any face n. Finally, there are only 4 unique interfaces, 1 for each interior face of an element.

In a 3-by-3 example of the problem we reconstruct $\mathbf{F}$ from the representational matrix, $\mathbf{F}^{\text{ind}}$, and the set of unique $\mathbf{F}^i$ sub-matrices, substituting the appropriate sub-matrix in place of the integer value, *i.e.*,

$$\mathbf{F} = \mathbf{F}^{\text{ind}}\left[i/\mathbf{F}^i\right] \text{ where } i \neq 0, \tag{8a}$$

$$\mathbf{F}^{\text{ind}} = \overset{\textit{Interface index}}{\begin{array}{c} \begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \\ \left[\begin{array}{cccccccccccc} 4 & \cdot & \cdot & \cdot & \cdot & \cdot & 2 & \cdot & \cdot & \cdot & \cdot & \cdot \\ 3 & 4 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & 3 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 2 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 4 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 2 & \cdot & \cdot \\ \cdot & \cdot & 3 & 4 & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 2 & \cdot \\ \cdot & \cdot & \cdot & 3 & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 2 \\ \cdot & \cdot & \cdot & \cdot & 4 & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 3 & 4 & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & 3 & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \end{array}\right] \end{array}} \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \; \textit{Element index} \tag{8b}$$

$$\left\{\mathbf{F}^1, \mathbf{F}^2, \mathbf{F}^3, \mathbf{F}^4\right\}. \tag{8c}$$

In this problem there are only 4 unique sub-matrices because the mesh of elements consists of identically sized 4-sided elements. The location of entries in each row of Eq. 8b correspond to the orientation of interfaces along a given element seen in Fig. 1, *e.g.*, element 2 has interfaces 1, 2, and 8, corresponding to non-zeros in the identical columns in row 2 of $\mathbf{F}^{\text{ind}}$.

3 METHODOLOGY

In this section, we analyze several operations in the problem and derive characteristics its performance given the size of each element, n^2, and number of elements ℓ^2. We implement our problem in a shared memory context via OpenMP [19] parallelism. As the techniques between shared memory and distributed memory parallelism vary, our description reflects the characteristics of this problem that benefit shared memory parallelism.

3.1 Performance model

With the decomposed matrices given by Eq. 8a–8c, and the analogous form for $\mathbf{M}$, we avoid storing identical sub-matrices, which a typical sparse matrix format

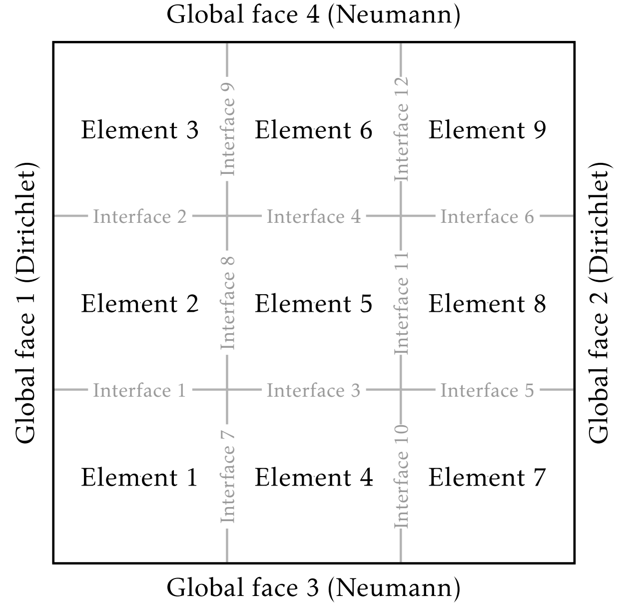

Figure 1. An illustration the volume, and arrangement of interfaces described by a 3-by-3 instance of the hybrid problem specified in (4).

would not. Additionally, where multiple operations utilize the same $\mathbf{F}^i\,\mathbf{M}_i$, we only perform this operation once. This holds for all sub-matrices of $\mathbf{F}^\intercal$ as each $(\mathbf{F}^\intercal)^i \equiv (\mathbf{F}^i)^\intercal$. Such operations are found in computing the global, trace problem *i.e.*, Eq. 6a–6c.

The benefit of this is now, each local system is computed as a series of smaller problems. This decomposition strategy is characteristic of hyrbidized methods, and is the fundamental reason to utilize these methods, in particular for larger problems sizes within a shared-memory context [10, 17, 18].

3.1.1 Complexity of the trace system

The hybrid formulation permits us to compute much larger problems with less memory and fewer reads, especially if the problem is mostly homogeneous. This is of particular interest as the 3 significant computations used in computing this problem *i.e.*, MatMul, SpMV, and linear solve via direct solve, are often memory-bound operations. Additionally, we reduce the computational complexity of solving the system by instead solving several smaller systems. In both cases this allows us to express the equations (6a–6c, 7) as a concatenation of decomposed, smaller problems.

To evaluate this the performance characteristics we fix the global number of volume points, $\bar{n}^2$, and vary the number of elements, to understand if there exists an optimal trade-off between the size of the decomposed matrices and quantity of computations derived from the decomposed matrices.

3.1.2 Computing the trace inverse product

In our implementation, the product $\mathbf{M}^{-1}\mathbf{F}$ in Eq. 6a is computed from a series a series of solves, storing $\mathbf{F}$ as a vector of vectors to compute $\mathbf{M}^{-1}\mathbf{F}$. That is, for each directional $\mathbf{F}^k$ boundary we store a list of n vectors

$$\mathbf{F}^k = \left[\mathbf{f}_1^k \cdots \mathbf{f}_n^k\right] \in \mathbb{R}^{n \times n^2}. \tag{9}$$

Here, each vector $\mathbf{f}_i^k$ is implemented as a PetscVector object. This is in contrast to storing $\mathbf{F}$ as a sparse matrix, and is necessary for the computation of $\mathbf{M}^{-1}\mathbf{F}$ from the local matrices of $\mathbf{M}$, stored similarly as

$$\mathbf{M} = \left\{\mathbf{M}_1 \cdots \mathbf{M}_p\right\} \in \mathbb{R}^{\ell^2 n^2 \times \ell^2 n^2}, p \leq \ell^2. \tag{10}$$

Where p denotes the number of unique local matrix operators in $\mathbf{M}$.

With this we compute the intermediate result $\mathbf{M}^{-1}\mathbf{F}$, performing a linear solve of every unique local matrix operator, and every vector of every directional boundary coefficient matrix

$$\text{solve}(\mathbf{M}_i, \mathbf{F}_j^k) \text{ for } \begin{array}{l} 0 < i \leq p, \\ 0 < j \leq n, \\ 0 < k \leq 4. \end{array} \tag{11}$$

This result has a similar non-zero pattern as $\mathbf{F}$, following the same symbolic block structure. In our implementation we solve this system through direct solvers made available through PETSc.

> In our isotropic 2-D Poisson equation in Eq. 1 we have
>
> $$p = \begin{cases} \ell & 0 \leq \ell < 3 \\ 3 & 3 \leq \ell \end{cases}. \tag{12}$$
>
> When $p = 3$ we have one unique $\mathbf{M}_i$ for the elements along the top Neumann boundaries, the bottom Neumann boundaries, and the interior elements. Since p is constant when $\ell \geq 3$, increasing the number of elements, ℓ^2 decreases the number of solves in Eq. 11, as well as the size of each solve, for a constant global volume, $\bar{n}^2$.

The number of solves we compute for $\mathbf{M}^{-1}\mathbf{F}^\intercal$ is then $p \times 3 \times n$. Thus the total FLOPS we use to compute elements of size $n^2 \times n^2$ using forwards/backwards substitution is

$$\text{FLOPS}(\mathbf{M}^{-1}\mathbf{F}^\intercal) = 24(\bar{n}/\ell)^5. \tag{13}$$

Since we fix the problem size $\bar{n}^2$, we write $n = \bar{n}/\ell$, allowing us to parameterize our model by the number of elements instead of the size of each element. As $\bar{n}$ is constant, computing $\mathbf{M}^{-1}\mathbf{F}^\intercal$ will require proportionally fewer FLOPS as the number of elements increases.

3.1.3 Computing the trace MatMul

The pattern of the $\mathbf{F}$ matrix is determined by the interfaces since we compute the product of each intermediary result in $\mathbf{M}^{-1}\mathbf{F}$ with the $\mathbf{F}^k$ interfaces that make a

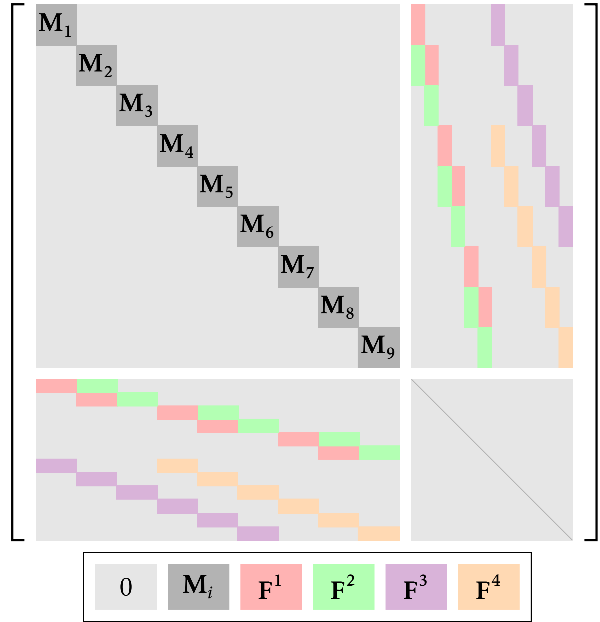

Figure 2. A visualization of the non-zero pattern of the matrix operator of a 3-by-3 instance of the hybrid problem specified in Eq. 4.

face in a given $\mathbf{M}_i$. This is illustrated by Fig. 2 where we see that each shaded row in $\mathbf{F}$ corresponds to an interface in Fig. 1. Again, we write the number of interfaces in terms of the number of elements, $r = 2\ell^2 - 2\ell$. The dimensions of $\boldsymbol{\lambda}_\mathbf{A}$ can then be written as

$$\boldsymbol{\lambda}_\mathbf{A} \in \mathbb{R}^{rn \times rn} \equiv \mathbb{R}^{2\bar{n}\ell - 2\bar{n} \times 2\bar{n}\ell - 2\bar{n}}. \tag{14}$$

The number of non-zeros in the intermediary result $\mathbf{F}^\intercal(\mathbf{M}^{-1}\mathbf{F})$ is given by

$$n^2 \sum_{i=1}^{\ell^2} (\phi_i^2) \tag{15a}$$

where

$$\mathbf{F}^{\text{ind}} = \left[\mathbf{f}_1^{\text{ind}} \cdots \mathbf{f}_{\ell^2}^{\text{ind}}\right] \in \mathbb{Z}^{\ell^2 \times r}, \tag{15b}$$

$$\phi = \left[\text{nnz}(\mathbf{f}_1^{\text{ind}}) \cdots \text{nnz}(\mathbf{f}_{\ell^2}^{\text{ind}})\right] \in \mathbb{Z}^{\ell^2}, \tag{15c}$$

or, the sum of the square of the number of internal interfaces per element. An example of the non-zero pattern of this result is shown in Fig. 3; the pattern of this matrix is identical to $\boldsymbol{\lambda}_\mathbf{A}$ as it just includes the addition $\mathbf{D}$. This sum demonstrates that $\boldsymbol{\lambda}_\mathbf{A}$ the number of non-zeros always increase as increase the number of elements.

The additional constraints we place on this problem, that being the square orientation of elements, we can write ϕ as

$$\phi = \Big[\underbrace{2, 2, 2, 2}_{4}, \underbrace{3 \cdots 3}_{4\ell - 8}, \underbrace{4 \cdots 4}_{(\ell-2)^2}\Big] \in \mathbb{Z}^{\ell^2}, \tag{16}$$

where 2's denote the corner elements, 3's denote the boundary elements, and 4's denote the internal elements. In Eq. 15a we square each ϕ_i because $\mathbf{F}^\intercal$ and

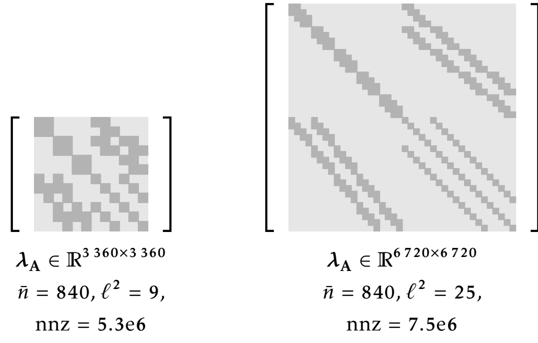

Figure 3. Additional non-zeros occur in $\lambda_{\mathbf{A}}$ when adding elements. For a constant number of grid points, the number of non-zero entries in this matrix increases when adding more elements.

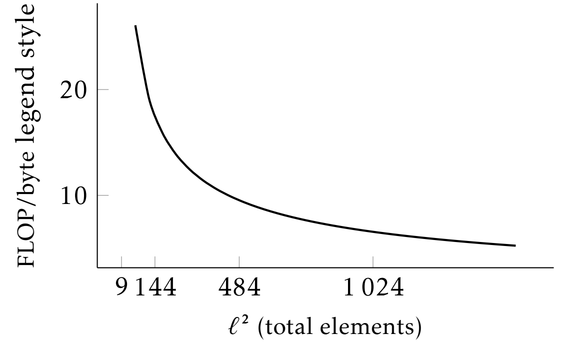

Figure 4. The arithmetic intensity (FLOPs/byte) of $\mathbf{F}^\intercal \times (\mathbf{M}^{-1}\mathbf{F})$.

$\mathbf{M}^{-1}\mathbf{F}$ can both be derived from substitutions of $\mathbf{F}^{\text{ind}}$, and the sum is multiplied by n^2 accounting for the additional dimension of each $\mathbf{F}^k$.

The ordering of the rows in $\mathbf{F}^{\text{ind}}$ may not be equivalent in this case, but the final sum will be identical. Throughout the rest of the problem this is not an issue if the ordering remains consistent.

From the number of non-zeros in $\lambda_{\mathbf{A}}$ we derive the total FLOPS to compute $\mathbf{F}^\intercal \times (\mathbf{M}^{-1}\mathbf{F})$ as a $v^\intercal v$ product of every n^2 length row in each $(\mathbf{F}^k)^\intercal$ and each n^2 length column in the intermediary matrices of $(\mathbf{M}^{-1}\mathbf{F})$. This is again, expressed as a function of the number of elements

$$\text{FLOPS}(\mathbf{F}^\intercal \times (\mathbf{M}^{-1}\mathbf{F})) = 2(\bar{n}/\ell)^4 \sum_{i=1}^{\ell^2} (\phi_i^2). \tag{17}$$

Here, we can also discretely determine the minimum number of bytes needed by this operation. That is, the memory needed for $\mathbf{F}^\intercal$, $(\mathbf{M}^{-1}\mathbf{F})$, and $\lambda_{\mathbf{A}}$

$$\text{BYTES}(\mathbf{F}^\intercal \times (\mathbf{M}^{-1}\mathbf{F})) = 8 \times 16(\bar{n}/\ell)^3 + 8 \times (\bar{n}/\ell)^2 \sum_{i=1}^{\ell^2} (\phi_i^2). \tag{18}$$

From this we derive the arithmetic intensity of this operation

$$\beta(\mathbf{F}^\intercal \times (\mathbf{M}^{-1}\mathbf{F})) = \frac{\text{FLOPS}(\mathbf{F}^\intercal \times (\mathbf{M}^{-1}\mathbf{F}))}{\text{BYTES}(\mathbf{F}^\intercal \times (\mathbf{M}^{-1}\mathbf{F}))} \tag{19}$$

which we plot as a function of the number of elements in Fig. 4.

3.1.4 Computing the trace vector

We utilize two potentially expensive kernels to compute $\lambda_{\mathbf{b}}$ in Eq. 6c, those being a matrix-vector linear solve, similar to the matrix-matrix solve in Eq. 6b, and an SpMV. The complexity of this is a solve is three polynomial orders lower than Eq. 13 at

$$\text{FLOPS}(\mathbf{M}^{-1}\bar{\mathbf{g}}) = \bar{n}^4/\ell^2. \tag{20}$$

This is evident as $\bar{\mathbf{g}}$ is a length $\bar{n}^2$ vector, not a matrix like $\mathbf{F}$; as $\bar{\mathbf{g}}$ is a unique vector for every $\mathbf{M}_i$ we compute ℓ^2 solves instead of $p \times 3 \times n$ solves.

To solve the SpMV we once again utilize the block diagonal structure of the problem to compute $\mathbf{F}^\intercal \times (\mathbf{M}^{-1}\bar{\mathbf{g}})$. We compute each component solution from each $\mathbf{F}^k$ to the appropriate intermediate vectors of $\mathbf{M}^{-1}\bar{\mathbf{g}}$. FLOPs are totaled through a sum similar to Eq. 15a, but only multiplied by n, as this is a matrix-vector operation. The total FLOPs to compute this operation is then

$$\text{FLOPS}(\mathbf{F}^\intercal \times (\mathbf{M}^{-1}\bar{\mathbf{g}})) = 12(\bar{n}/\ell) \sum_{i=1}^{\ell^2} (\phi_i^2). \tag{21}$$

This appears as a 1st order polynomial because each $\mathbf{F}^k$ contains $6(\bar{n}/\ell)$ entries is used only once for each entry in $\mathbf{F}^{\text{ind}}$. We also then have the number of bytes needed by SpMV in this instance

$$\text{BYTES}(\mathbf{F}^\intercal \times (\mathbf{M}^{-1}\bar{\mathbf{g}})) = 96(\bar{n}/\ell) \sum_{i=1}^{\ell^2} (\phi_i^2). \tag{22}$$

3.1.5 Computing the trace solve

To solve $\lambda_{\mathbf{A}}$ we use a direct Cholesky solver provided through PETSc. Here the total FLOPs to solve a $2\bar{n}\ell - 2\bar{n} \times 2\bar{n}\ell - 2\bar{n}$ system is

$$\text{FLOPS}(\lambda_{\mathbf{A}}^{-1}\lambda_{\mathbf{b}}) = 2(2\bar{n}\ell - 2\bar{n})^2. \tag{23}$$

3.1.6 Computing the local solves

Finally, to compute the solution via the local systems in PETSc's Cholesky solver in Eq. 7 we have ℓ, $n^2 \times n^2$ component matrices of $\mathbf{M}$, by ℓ sub-sections of the vector $(\bar{\mathbf{g}} - \mathbf{F}\lambda)$. The total FLOPs are then

$$\text{FLOPS}(\mathbf{M}^{-1}(\bar{\mathbf{g}} - \mathbf{F}\lambda)) = 2(\bar{n}^4/\ell^5). \tag{24}$$

4 EXPERIMENTS

In this section we complement our analysis with performance results of our parallel implementation of

Threads	Elements (ℓ^2)																
	9	16	25	36	49	64	100	144	196	225	400	441	576	784	900	1225	1600
1	490	330	250	200	180	160	160	160	170	180	220	230	260	320	360	490	720
2	270	200	150	110	89	81	75	77	80	82	100	110	120	160	170	230	350
4	150	110	83	64	48	43	40	41	43	45	57	58	66	81	92	130	190
8	89	68	53	38	30	26	24	26	24	25	31	31	38	45	50	67	97
16	59	39	31	22	18	15	15	13	14	15	16	17	20	23	26	35	51
28	45	30	20	15	12	9.7	8.4	8.5	8.7	9.1	11	11	12	14	16	21	29

Figure 1. Compute time (s) given a fixed problem size for various element and thread configurations on a single Xeon E5-2690 node.

the 2D hybridized SBP-SAT problem. We developed code used to solve this system in C++ and utilize the PETSc library to implement the linear algebra operations utilized throughout. All linear systems are solved using the direct LU solver provided by PETSc. We implemented thread parallel routines in OpenMP [19], utilizing the parallelism made available through hybridization. Our code ran on the University of Oregon's Talapas supercomputing cluster on an Intel Xeon E5-2690 v4 CPU. Profiling information including cache accesses, writes, and flop counts were sought using the PAPI profiler [16].

4.1 Strong scaling

We ran several problems with an fixed number of grid points in the problem's volume, varying n and ℓ proportionally, *i.e.*, $\bar{n} = n \times \ell$ for some fixed ℓ. Each hybridized system has more than 705 600 points as the $\mathbf{F}$ factors increase the size of the system as we add elements. Each problem was run with several thread configurations, from 1 thread to 28 threads. Additionally total number of elements varied between 9, with 78 400 grid points per element and 1600, with 441 grid points per elements. With this we have a profile of the problem's strong-scaling performance, as well as an insight in the effects of varying element size within the same problem.

The optimal runtime was found at $\ell^2 = 100$ for all thread configurations. A full table of the compute time separated by thread count and element count is given in Fig. 1. We plot the speedup *i.e.*, $S = T/T_p$ for the number of threads p for the best and worst choices of ℓ in Fig. 5.

Here, speedup scaling is proportional to ℓ. This is not in proportion to the overall runtime by choice of ℓ; the best case ($\ell^2 = 100$) has worse scaling than $\ell^2 = 1600$ and better than $\ell^2 = 9$. This suggests that different components of the problem are contributing to the performance on either end of of this scale.

To this end we profiled the runtime of the 7 operations that comprise the hybridized system. Two operations utilize the majority of the compute time, those being SpMV from Eq. 6b and MatMul from Eq. 6a. The remaining 5 operations average 8% of the compute time when run on a single thread. The compute time of the two major operations are plotted in Fig. 6 with the same configuration of problems.

We exclude the time spent factorizing $\mathbf{M}$ and $\lambda_{\mathbf{A}}$ for this problem, its practical application would likely involve the reuse of these factorization several times. It is worth noting that the cost of LU factorization for $\lambda_{\mathbf{A}}$ becomes considerable as ℓ^2 increases. This is expected by Eq. 14 as the size of $\lambda_{\mathbf{A}}$ increases with ℓ and by Eq. 15a as the number of non-zeros increases with ℓ.

4.2 MatMul analysis

Out of the two kernels that comprise the majority of the runtime we see the most variability in MatMul operation from Eq. 6a. The operation is implemented in a similar manner to batched general matrix multiply (GEMM) methods [24], but as our intermediate results from $\mathbf{M}^{-1}\mathbf{F}$ are dense vectors we instead compute a batched vector inner product. Batching methods allocate single threads to self-contained operations in contrast to of coordinating multiple threads to solve a larger problem.

These results show an optimal range of elements to minimize runtime between 100 to 600 elements for $\bar{n} = 840$. When there are too few elements the problem has a greater number of FLOPs and bytes loaded. The total FLOPs of this operation decrease for MatMul at a rate of $1/\ell^4$ from Eq. 17 and the total bytes decrease at $1/\ell^3$ from Eq. 18. As both of these terms decrease, our runtime decreases proportionally until we reach approximately 600 elements.

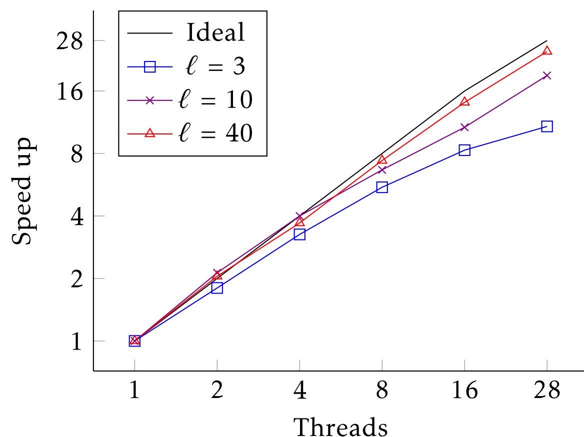

Figure 5. The speed up of three pairs of n and ℓ such that $\bar{n}$ is constant. Scaling improves as ℓ increases; however, overall runtime is best at $ell = 10$.

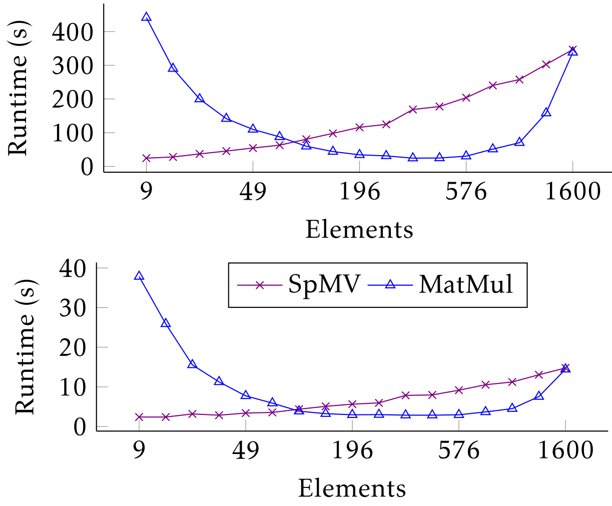

Figure 6. Runtime of routines that significantly contribute to the overall runtime for various element configurations. 1 thread (top) and 28 threads (bottom).

4.3 SpMV analysis

The SpMV kernel is similarly batched to compute $\mathbf{F}^\mathsf{T} \times (\mathbf{M}_i^{-1}\bar{\mathbf{g}})$ for each element. Unlike MatMul, each intermediate vector in $\mathbf{M}_i^{-1}\bar{\mathbf{g}}$ is unique since $\bar{\mathbf{g}}$ is unique. Additionally, we compute proportionally fewer batches of $n^2 \times n$ SpMV than MatMul, and write out to significantly fewer memory locations. We still encounter the same write miss penalty as we both increase the size of the intermediate vector $\mathbf{F}^\mathsf{T} \times (\mathbf{M}_i^{-1}\bar{\mathbf{g}})$ and decrease the size of each batch operation proportionally. Even at the smallest element size each batch write 21 doubles, which is larger than a typical cache line.

4.4 Memory behavior

To investigate the decreasing performance of our MatMul and SpMV operations we chose pairs of $\bar{n}$ and ℓ with similar total flops for the MatMul kernel. These problems averaged 1.32e11 flops with a coefficient of

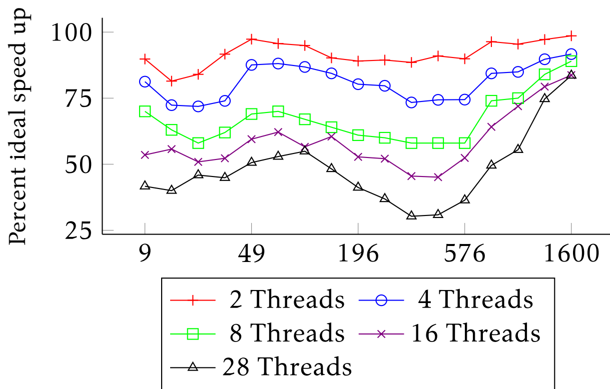

Figure 7. Performance of the decomposed MatMult kernel on a fixed problem size (705600 grid points) for various thread counts and total elements.

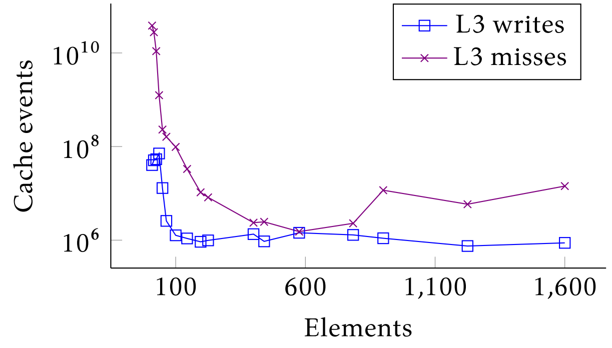

Figure 8. L3 writes and misses profiled for the strong scaling problem configuration. Though the total bytes written out to $\lambda_\mathbf{A}$ increase with additional elements, fewer occur with every batch. Fewer writes per batch to more rows of $\lambda_\mathbf{A}$ increase our cache misses and overall runtime.

variation of 2.3e−3. Though the flops between each problem varies insignificantly, and the bytes loaded *decrease*, the number of entries in $\lambda_\mathbf{A}$ — and hence the number of bytes written *increases*. This is illustrated in on the left of Fig. 9; this also plots the measured number of L3 cache writes in MatMul kernel on the right. The number of L3 writes grows proportional to the number of entries in $\lambda_\mathbf{A}$. The number of entries are approximately 80× greater than the number of L3 writes. This is slightly less than the number of 8 byte doubles that fit into a 128 byte cache line.

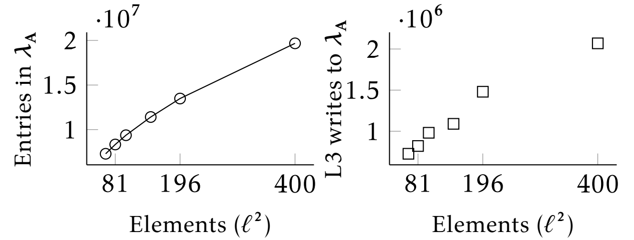

Figure 9. The amount of non-zero elements in $\lambda_\mathbf{A}$ for a set of problems with similar flop profiles (left) and the number of L3 cache data writes measured with PAPI (right). While the total flops are similar, the size and number of entries in $\lambda_\mathbf{A}$ continues to grow.

The runtime of MatMul SpMV, and the LU factorization of $\lambda_\mathbf{A}$ all increase in runtime, even though the flops remaining minimally changed and the bytes loaded decrease. We illustrate this in Fig. 10. The growth of the $\lambda_\mathbf{A}$ factorization ide to the additional non-zero entries increasing its complexity. For the other two operations this suggests that the greatest impact to runtime at this scale is the number of writes to $\lambda_\mathbf{A}$, *i.e.*, that these operations are becoming write bound. This is affirmed by Fig. 8, as additional the sum of L3 write and misses in the MatMul kernel as we increase the number of elements while fixing grid points.

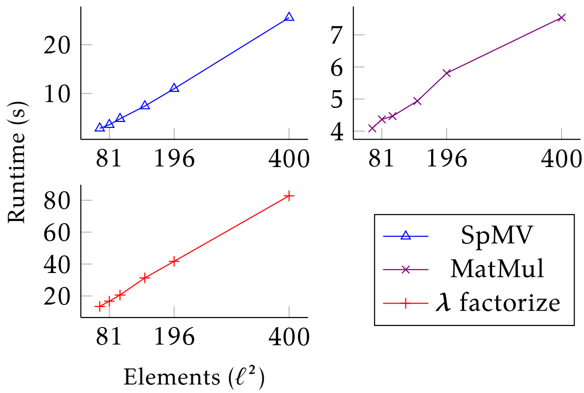

Figure 10. The runtime of three expensive operations using 28 threads on several problem sizes with a similar total flops. Though the total flops remain essentially the same, the runtime increases for all three kernels.

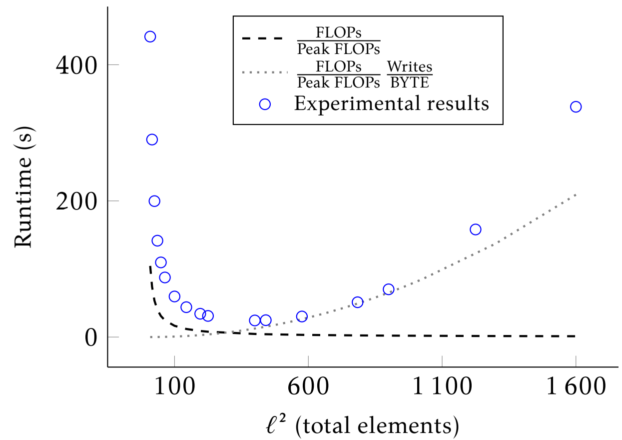

Figure 11. MatMul performance models (s/ℓ^2) plotted against single-threaded results.

5 DISCUSSION

The results of our implementation of the hybridized method and subsequent experiments show that the bulk of the problem's compute time is spent setting up the hybridized method. Solving all local problems requires at most 1.5% of the total compute time. The remaining 98.5% of the compute time is spent solving the global system, and 91% of the compute time is spend solely on the MatMul and SpMV operations.

5.1 Selecting an optimal batch size

All major operations in this problem are thread parallel batch operations, which often effectively map to the mathematics of the hybridized method. We chose a problem that could be batched as evenly as possible to access the most parallelism in this model. We demonstrated that the two expensive operations have optimal performance ranges. For the MatMul operation this occurs when we decrease the amount of work, but are not yet write bound. We plot time estimates from these two metrics in Fig. 11. The FLOPs model estimates time from the FLOPs function in Eq. 17 normalized by the system's peak single core FLOPs (8e+9). The memory model is identical to the FLOPs model, but also multiplies the ratio of writes to bytes loads. In this model we optimize for both the minimal number of flops, and the minimal write to load ratio.

We should be particularly interested in minimizing the write to load ratio, as it also resembles the increasing the runtime in the SpMV operation.

5.2 Optimizing MatMul

Our implementation of batched MatMul naively computes batches, looping over each row and column of the components of $\mathbf{F}^\intercal$ and $\mathbf{M}^{-1}\mathbf{F}$ to compute $\lambda_\mathbf{A}$. This implementation doesn't translate to ideal cache utilization and is for our purposes, random. Recall that the ordering of $\mathbf{F}^{\text{ind}}$ does not matter as long as it remains consistent throughout the problem. We could attempt to achieve a more ideal cache utilization, writing to $\lambda_\mathbf{A}$ row by row as much as possible. This could also be achieved without reordering $\mathbf{F}^{\text{ind}}$ by modifying the underlying batching algorithm.

At scales where the problem and the work is larger, such as where $\ell < 10$ in our profiling problem there is still room to optimize there problems, as suggested by the FLOPs model Fig. 11. These problems minimize the runtime on the other operations of the global system, but increase the runtime of computing the global system. We implemented these operations with PETSc, but other highly optimized libraries such Intel MKL may get us closer to the theoretical peak FLOPs. Such problems would inherently be limited to isotropic, but still applicable for large systems.

The best performance of our problem ($\ell^2 = 100$) approaches the theoretical peak FLOPs, but scales poorly. This suggests that while the problem utilizes the CPU effectively on a single thread, it is encountering additional cache misses that are being hidden by an ideal balance in work to cache latency. As we add threads to the problem it computes more batches in the same amount of time, but that time is still bound by the cache latency. Further, as we reduce the arithmetic intensity by adding elements, we again reveal the cache latency, but in this case we have also added additional writes.

A solution here might use a specialized sparse block matrix formats. To implement $\mathbf{F}$ we on store the matrix components, instead of storing it outright. A similar representation would reduce the work and the number of writes to $\lambda_\mathbf{A}$, but could cause more caching issues either when factorizing $\lambda_\mathbf{A}$ or using it in an iterative method.

5.3 Future work

This work largely profiled a single problem with a constant volume, but substituting different element sizes in each mesh. Interesting problems that are larger,

with more variability, and complex meshes require increasingly detailed models of performance. Recent work that approaches larger anistropic problems with complex meshes often has utilizes additional resources such as machine learning [14, 20] or functional analysis Ramabathiran and Ramachandran [20]. Another work from Hutter and Solomonik [15] utilizes tensor completion methods to construct performance models. A similar method could utilize the parameters of this problem, including the number of elements, the size of the problem, the variability in the domain, and the shape of the grid to construct performance models for large problems with performance data from small problems.

6 SUMMARY

In this work we presented an implementation of a Hybridized 2-D Poisson equation, implemented in the PETSc scientific computing library, and parallelized in OpenMP. Hybridized problems are assembled from multiple elements and coupled with interface conditions, which in total comprises the problem's volume. This problem is solved in two parts, wherein a sparse system is solved, allowing for the solution to be computed independently for each element. We developed a theoretical model of the work required to compute the problem in FLOPs, derived from the mathematics of the problem. We also provided a description of the problem size in bytes loaded, and the number of write operations that occur in major operations. We conducted an experiment to demonstrate that the overall time-to-solution for this system is heavily effected by the number and size of each element chosen. For one problem with 705 600 grid points we found the optimal runtime when we used 100 40×40 elements. Finally, we determined that the largest operations in the problem were latency bound, discussed strategies for reducing the effect, and what effects this might have on larger problems.

References

[1] Karrar Kadum Abbas and Xianping Li. Anisotropic mesh adaptation for image segmentation based on mumford-shah functional. *ArXiv*, abs/2007.08696, 2020. URL https://api.semanticscholar.org/CorpusID:220633259.

[2] Vahid Badrkhani, Rene R Hiemstra, Michal Mika, and Dominik Schillinger. A matrix-free macro-element variant of the hybridized discontinuous galerkin method. *arXiv preprint arXiv:2302.10917*, 2023.

[3] Satish Balay, Shrirang Abhyankar, Steven Benson, Jed Brown, Peter R Brune, Kristopher R Buschelman, Emil Constantinescu, Alp Dener, Jacob Faibussowitsch, William D Gropp, et al. Petsc/tao users manual. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2022.

[4] Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of computational Physics*, 53(3):484–512, 1984.

[5] Alex Bespalov, David J Silvester, and Feng Xu. Error estimation and adaptivity for stochastic collocation finite elements part i: single-level approximation. *SIAM Journal on Scientific Computing*, 44 (5):A3393–A3412, 2022.

[6] Matthias Bollhöfer, Olaf Schenk, Radim Janalik, Steve Hamm, and Kiran Gullapalli. State-of-the-art sparse direct solvers. *Parallel algorithms in computational science and engineering*, pages 3–33, 2020.

[7] Alexandre Chen. Iterative methods for earthquake cycle simulations with hpc applications.

[8] Joseph P Dean, Sander Rhebergen, and Garth N Wells. Design and analysis of a hybridized discontinuous galerkin method for incompressible flows on meshes with quadrilateral cells. *arXiv preprint arXiv:2306.05288*, 2023.

[9] Alain Dervieux, David Leservoisier, Paul-Louis George, and Yves Coudiere. About theoretical and practical impact of mesh adaptation on approximation of functions and pde solutions. *International journal for numerical methods in fluids*, 43(5):507–516, 2003.

[10] Pablo Fernandez, Ngoc Cuong Nguyen, and Jaime Peraire. The hybridized discontinuous galerkin method for implicit large-eddy simulation of transitional turbulent flows. *Journal of Computational Physics*, 336:308–329, 2017.

[11] Matteo Frigo, Nicola Castelletto, Massimiliano Ferronato, and Joshua A White. Efficient solvers for hybridized three-field mixed finite element coupled poromechanics. *Computers & Mathematics with Applications*, 91:36–52, 2021.

[12] Caroline Geiersbach and Winnifried Wollner. A stochastic gradient method with mesh refinement for pde-constrained optimization under uncertainty. *SIAM Journal on Scientific Computing*, 42(5):A2750–A2772, 2020.

[13] Robert J Guyan. Reduction of stiffness and mass matrices. *AIAA journal*, 3(2):380–380, 1965.

[14] Ru Huang, Ruipeng Li, and Yuanzhe Xi. Learning optimal multigrid smoothers via neural networks. *ArXiv*, abs/2102.12071, 2021. URL https://api.semanticscholar.org/CorpusID:232035769.

[15] Edward Hutter and Edgar Solomonik. Application performance modeling via tensor completion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701092. doi: 10.1145/3581784.3607069. URL https://doi.org/10.1145/3581784.3607069.

[16] Heike Jagode, Anthony Danalis, Hartwig Anzt, and Jack Dongarra. Papi software-defined events for in-depth performance analysis. *The International Journal of High Performance Computing Applications*, 33(6):1113–1127, 2019.

[17] Tzanio Kolev, Paul Fischer, Misun Min, Jack Dongarra, Jed Brown, Veselin Dobrev, Tim Warburton, Stanimire Tomov, Mark S Shephard, Ahmad Abdelfattah, et al. Efficient exascale discretizations: High-order finite element methods. *The International Journal of High Performance Computing Applications*, 35(6):527–552, 2021.

[18] Jeremy E Kozdon, Brittany A Erickson, and Lucas C Wilcox. Hybridized summation-by-parts finite difference methods. *Journal of Scientific Computing*, 87(3):85, 2021.

[19] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008. URL http://www.openmp.org/mp-documents/spec30.pdf.

[20] Amuthan Arunkumar Ramabathiran and Prabhu Ramachandran. Anisotropic, sparse and interpretable physics-informed neural networks for pdes. *ArXiv*, abs/2207.00377, 2022. URL https://api.semanticscholar.org/CorpusID:250243921.

[21] Sander Rhebergen and Garth N Wells. Analysis of a hybridized/interface stabilized finite element method for the stokes equations. *SIAM Journal on Numerical Analysis*, 55(4):1982–2003, 2017.

[22] Ashesh Sharma, Shreyas Ananthan, Jayanarayanan Sitaraman, Stephen Thomas, and Michael A Sprague. Overset meshes for incompressible flows: On preserving accuracy of underlying discretizations. *Journal of Computational Physics*, 428:109987, 2021.

[23] Rüdiger Verfürth. A posteriori error estimation and adaptive mesh-refinement techniques. *Journal of Computational and Applied Mathematics*, 50(1-3):67–83, 1994.

[24] Cunyang Wei, Haipeng Jia, Yunquan Zhang, Kun Li, and Luhan Wang. Lbbgemm: A load-balanced batch gemm framework on arm cpu s. In *2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*, pages 59–66. IEEE, 2022.

[25] Edward L Wilson. The static condensation algorithm. *International Journal for Numerical Methods in Engineering*, 8(1):198–203, 1974.