AN ADAPTIVE AND PARALLEL DIRECT SOLVER FOR ELLIPTIC PARTIAL DIFFERENTIAL EQUATIONS

by

Damyn M. Chipman

A dissertation

submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy in Computing

Boise State University

May 2024

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Damyn M. Chipman

Thesis Title: An Adaptive and Parallel Direct Solver for Elliptic Partial Differential Equations

Date of Final Oral Examination: 25 March 2024

The following individuals read and discussed the dissertation submitted by student Damyn M. Chipman, and they evaluated the student's presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Donna Calhoun Ph.D.	Chair, Supervisory Committee
Michal Kopera Ph.D.	Member, Supervisory Committee
Grady Wright Ph.D.	Member, Supervisory Committee
Kirk Ketelsen Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Donna Calhoun Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

TABLE OF CONTENTS

LIST OF FIGURES ix

LIST OF TABLES xiii

LIST OF ABBREVIATIONS xvi

1 INTRODUCTION AND OVERVIEW 1

1.1 Introduction 1

1.2 Literature Review 2

1.2.1 2

1.3 Novelty of this Work 2

1.4 Overview 2

2 ELLIPTIC PDES AND CLASSIC SOLUTION METHODS 3

2.1 The Elliptic Partial Differential Equation 3

2.2 Numerical Methods for Elliptic PDEs 5

2.2.1 Finite Difference 5

2.2.2 Finite Volume 8

2.2.3 Finite Element 9

2.2.4 Spectral Methods 10

2.2.5 Other Methods and Summary 11

2.3 Solution Methods for Elliptic Partial Differential Equations 12
2.3.1 Iterative Methods 12
2.3.2 Direct Methods 15
2.3.3 Hierarchical Methods 16

3 THE QUADTREE-ADAPTIVE HPS METHOD 23
3.1 Introduction 23
3.1.1 Problem Statement 26
3.2 Components of the HPS algorithm : Merging, Splitting and Leaf Computations 27
3.2.1 Leaf Level Computations 27
3.2.2 The 4-to-1 Merge Algorithm 30
3.2.3 Coarse-Fine Interfaces 38
3.2.4 Comparison Between 4-to-1 Merging and 2-to-1 Merging . . . 40
3.3 Implementation Details 41
3.3.1 Patch Solvers 41
3.3.2 Building Leaf Level Operators 42
3.3.3 Quadtrees and Adaptive Meshes 43
3.4 Numerical Results 47
3.4.1 Poisson Equation 1 47
3.4.2 Poisson Equation 2 (Polar-Star Problem) 50
3.4.3 Helmholtz's Equation 53
3.5 Conclusion 60

4 AN ADAPTIVE REBUILD IMPROVEMENT TO THE QUADTREE-ADAPTIVE

HPS METHOD 62
4.1 Introduction 62
4.2 Mathematical Theory for the Adaptive Rebuild 62
4.3 The Adaptive-Rebuild Algorithm 62
4.4 Numerical Experiments 62
4.5 Conclusion 62

5 A PARALLEL IMPLEMENTATION OF THE QUADTREE-ADAPTIVE HPS METHOD 63
5.1 Introduction 63
5.2 Overview of the Quadtree-Adaptive HPS Method 66
5.2.1 Problem Statement and Domain Representation 66
5.2.2 Building the Set of Solution Operators 68
5.2.3 Stages of the Quadtree-Adaptive HPS Method 72
5.3 The Parallel Algorithm 75
5.3.1 MPI Preliminaries 75
5.3.2 Quadtree Data Structures 76
5.4 Parallel Results and Discussion 80
5.4.1 Strong Scaling 80
5.4.2 Weak Scaling 82
5.5 Conclusion 86

6 APPLICATIONS OF THE HPS METHOD TO COUPLED SYSTEMS OF ELLIPTIC PDES 88

7 CONCLUSION 89

REFERENCES 89

APPENDICES 97

A RELEVANT SOFTWARE 98

A.1 EllipticForest 98

A.1.1 Summary 98

A.1.2 Statement of Need 99

A.1.3 Software Overview 101

LIST OF FIGURES

2.1 Discretization Methods. Top Left: Finite difference grid based on a mesh of points or nodes. Top Right: Finite volume mesh made of cells (dashed blue boxes) with cell averages in the center (red points). Bottom Left: Finite element mesh where each blue box is an element with a shape function defined at all points within the element. Bottom Right: Spectral mesh made up of a tensor of Chebyshev nodes. 6

2.2 The 1D Multigrid Method. Starting with the finest grid, perform a few iterations of an iterative method. This is called relaxing the solution. Then project onto a coarser grid, and relax again. Do this down the levels in the grid to a desired precision. Once the solution is converged to on the coarsest level, interpolate back up the levels to obtain the solution on the finest level. 18

2.3 HPS Merge Operation. The merged patch Ω_τ is the union of children Ω_α and Ω_β, i.e. $\Omega_\tau = \Omega_\alpha \cup \Omega_\beta$. Red, green, and blue nodes correspond to index sets $\mathbf{I}_1$, $\mathbf{I}_2$, and $\mathbf{I}_3$, respectively. The merge operation eliminates the nodes on the interface of the children patches. 21

2.4 HPS Solve Stage. Once $\mathbf{S}_0$ is formed, apply it to the top level Dirichlet data to get boundary (and solution) data on the interface of the children. Apply the patch solution operator down the tree until each leaf has it's local boundary information. Then apply the solution operator to get the solution data in the interior of each leaf. 22

3.1 The 4-to-1 merge process: (left) the four children patches with their local grid, (middle) the four children share internal and external boundaries with τ, (right) the merged parent patch with data on the exterior of τ. 31

3.2 Example of a coarse-fine interface merge: (left) patches $6, 7, 8, \& \ 9$ will be merged and result in a coarse-fine interface, (middle) the data on patch 5 is averaged (coarsened), (right) merging $2, 3, 4, \& \ 5$ can continue as detailed in Section 3.2.2. 39

3.3 Leaf-indexed vs. path-indexed quadtrees. In (a), only the leaves of a quadtree are indexed and stored. In (b), all nodes of the quadtree are indexed and stored according to their unique path. 44

3.4 A logically square domain is recursively broken into 4 children domains. This mesh represents refinement criteria that refines the center of the domain. The leaf level nodes are indexed according to a leaf-indexed quadtree and follow a space filling curve. 45

3.5 The computed solution and mesh for the Poisson problem Section 3.4.1. Patch size for this plot is 16×16 and mesh is refined to level 7. Refinement criteria is based on the magnitude of the right hand side function $f(x, y)$. 48

3.6 The computed solution and mesh for the Polar Star Poisson problem. Each patch has a 16×16 cell-centered grid. The mesh is refined according to the right-hand side and is refined with 8 levels of refinement. 52

3.7 Mesh and plot of the solution to the Helmholtz problem in Section 3.4.3. 56

3.8 Mesh and right-hand side of the solution to the Helmholtz problem in Section 3.4.3. 57

5.1 An example of an adaptive, quadtree mesh that is refined around the center of the domain. The colors denote different ranks and the numbers indicate the leaf-index of the quadrant. 68

5.2 Leaf-indexed vs. path-indexed quadtrees. Both trees represent the mesh found in Figure 5.1. The colors denote which rank owns each node. In (a), only the leaves of a quadtree are indexed and stored. In (b), all nodes of the quadtree are indexed and stored according to their unique path. Note that the nodes in (b) that are colored with a gradient (i.e., "0", "01", "02", "021") are owned by multiple ranks. . 69

5.3 The strong scaling plots for the (a) build stage, (b) upwards stage, and (c) solve stage. For each, the left plot shows the scaling for the leaf callback and the right plot shows the scaling for the family callback. The solid line indicates actual timing and the dashed line indicates ideal strong scaling. 83

5.4 Plots of the polar star Poisson problem used in the weak scaling study found in Section 5.4.2. The polar star is generated from the RHS of the Poisson equation and is repeated for each MPI rank used to solve the problem. This shows four polar stars arranged in a 2×2 grid for solving with $N_R = 4$. Each outlined grid contains a 16×16 finite volume mesh. 85

5.5 The weak scaling plots for the (a) build stage, (b) upwards stage, and (c) solve stage. For each, the left plot shows the scaling for the leaf callback and the right plot shows the scaling for the family callback. The solid line indicates actual efficiency and the dashed line indicates ideal weak scaling. 87

A.1 A path-indexed quadtree representation of a mesh. Colors indicate which rank owns that node. The nodes colored by gradient indicate they are owned by multiple ranks. 103

A.2 Solution of Poisson equation on a quadtree mesh using EllipticForest. The mesh and data are output in an unstructured PVTK format and visualized with VisIt **?**. 106

LIST OF TABLES

2.1 Iterative Methods: Splitting Methods 13

2.2 Iterative Methods: Krylov Subspace Methods 14

3.1 Convergence analysis for Poisson's equation. The upper part shows convergence for a uniformly refined mesh, while the lower part shows convergence for an adaptively refined mesh. M is the size of the grid on each leaf patch, $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution for a uniformly refined mesh, DOFs is the total degrees of freedom (i.e., total mesh points), L_∞ error is the infinity norm error, L_∞ order is the infinity norm convergence order, L_1 error is the 1^{st} norm error, and L_1 order is the 1^{st} norm convergence order. 49

3.2 Timing and memory results for Poisson's equation. The upper part shows results for the uniformly refined mesh, while the lower part shows results for the adaptively refined mesh. The results here are for a patch size of 16×16. $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution, DOFs is the total degrees of freedom, T_{build} is the time in seconds for the build stage, T_{upwards} is the time in seconds for the upwards stage, T_{solve} is the time in seconds for the solve stage, and S is the memory storage in megabytes to store the quadtree and all data matrices stored in each node of the quadtree. 50

3.3 Polar Star Poisson Problem Parameters 53

3.4 Convergence analysis for the Polar Star Poisson problem. The upper part shows convergence for a uniformly refined mesh, while the lower part shows convergence for an adaptively refined mesh. M is the size of the grid on each leaf patch, $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution for a uniformly refined mesh, DOFs is the total degrees of freedom (i.e., total mesh points), L_∞ error is the infinity norm error, L_∞ order is the infinity norm convergence order, L_1 error is the 1^{st} norm error, and L_1 order is the 1^{st} norm convergence order. 54

3.5 Timing and memory results for the Polar Star Poisson problem. The upper part shows results for the uniformly refined mesh, while the lower part shows results for the adaptively refined mesh. The results here are for a patch size of 16×16. $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution, DOFs is the total degrees of freedom, T_{build} is the time in seconds for the build stage, T_{upwards} is the time in seconds for the upwards stage, T_{solve} is the time in seconds for the solve stage, and S is the memory storage in megabytes to store the quadtree and all data matrices stored in each node of the quadtree. 55

3.6 Convergence analysis for the Helmholtz problem. The upper part shows convergence for a uniformly refined mesh, while the lower part shows convergence for an adaptively refined mesh. M is the size of the grid on each leaf patch, $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution for a uniformly refined mesh, DOFs is the total degrees of freedom (i.e., total mesh points), L_∞ error is the infinity norm error, L_∞ order is the infinity norm convergence order, L_1 error is the 1^{st} norm error, and L_1 order is the 1^{st} norm convergence order. 58

3.7 Timing and memory results for the Helmholtz problem. The upper part shows results for the uniformly refined mesh, while the lower part shows results for the adaptively refined mesh. The results here are for a patch size of 16×16. $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution, DOFs is the total degrees of freedom, T_{build} is the time in seconds for the build stage, T_{upwards} is the time in seconds for the upwards stage, T_{solve} is the time in seconds for the solve stage, and S is the memory storage in megabytes to store the quadtree and all data matrices stored in each node of the quadtree. 59

LIST OF ABBREVIATIONS

This document is incomplete. The external file associated with the glossary 'acronym' (which should be called `BSUmain.acr`) hasn't been created.

Check the contents of the file `BSUmain.acn`. If it's empty, that means you haven't indexed any of your entries in this glossary (using commands like `\gls` or `\glsadd`) so this list can't be generated. If the file isn't empty, the document build process hasn't been completed.

Try one of the following:

- Add `automake` to your package option list when you load `glossaries-extra.sty`. For example:

  ```
  \usepackage[automake]{glossaries-extra}
  ```

- Run the external (Lua) application:

  ```
  makeglossaries-lite.lua "BSUmain"
  ```

- Run the external (Perl) application:

  ```
  makeglossaries "BSUmain"
  ```

Then rerun LaTeX on this document.

This message will be removed once the problem has been fixed.

CHAPTER 1:

INTRODUCTION AND OVERVIEW

1.1 Introduction

Elliptic partial differential equations describe physical systems that arise in the areas of physics, biology, chemistry, engineering, and others. They model phenomena like heat and mass dispersion, electromagnetism, fluid systems, and more. Modern approaches to solving such elliptic partial differential equations (PDEs) include numerical methods that target computational resources that range from a personal laptop or desktop computer to clusters and the largest supercomputers. Making efficient use of massive compute power is vital to make advances in scientific and engineering fields.

The numerical methods employed to solve PDEs on modern platforms are non-trivial in their algorithms or implementation. Developing software that targets CPU or GPU machines

Depending on the application, it is advantageous to use techniques that improve the accuracy of the numerical solution but come at the cost of algorithmic or software complexity. Such techniques include adaptive mesh refinement (AMR), fast methods for solving the subsequent linear systems, or a variety of discretization methods.

1.2 Literature Review

1.2.1

1.3 Novelty of this Work

1.4 Overview

This document will continue as follows:

- asdf
- asdf
- asdf

CHAPTER 2:

ELLIPTIC PDES AND CLASSIC SOLUTION METHODS

2.1 The Elliptic Partial Differential Equation

Partial differential equations are classified by their highest order derivative terms. A second order, linear differential equation can be written as

$$A(\mathbf{x})\frac{\partial^2 u(\mathbf{x})}{\partial x^2} + B(\mathbf{x})\frac{\partial^2 u(\mathbf{x})}{\partial x \partial y} + C(\mathbf{x})\frac{\partial^2 u(\mathbf{x})}{\partial y^2} + ...$$
$$... + D(\mathbf{x})\frac{\partial u(\mathbf{x})}{\partial x} + E(\mathbf{x})\frac{\partial u(\mathbf{x})}{\partial y} + F(\mathbf{x})u(\mathbf{x}) + G(\mathbf{x}) = 0.$$

Second-order linear PDEs are classified according to the value of the determinant of this expression:

$$B^2 - 4AC < 0, \quad \text{Elliptic}$$
$$B^2 - 4AC = 0, \quad \text{Parabolic}$$
$$B^2 - 4AC > 0, \quad \text{Hyperbolic}$$

In this overview, we will consider common elliptic PDEs such as the Poisson

equation

$$\nabla \cdot (\beta(\mathbf{x})\nabla u(\mathbf{x})) = f(\mathbf{x}), \tag{2.1}$$

and the Helmholtz equation

$$\nabla \cdot (\beta(\mathbf{x})\nabla u(\mathbf{x})) + \lambda(\mathbf{x})u(\mathbf{x}) = f(\mathbf{x}), \tag{2.2}$$

where $\mathbf{x} = [x, y]$, $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y})$, $\nabla^2 = \nabla \cdot \nabla = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$, and $\mathbf{x} \in \Omega \subset \mathcal{R}^2$. When $\beta(\mathbf{x}) = 0$ and $\lambda(\mathbf{x}) = \kappa^2$ (κ is a constant), these expressions reduce to the more classical, constant coefficient versions of the Poisson and Helmholtz equations. When the right-hand side function $f(\mathbf{x}) = 0$, these are homogeneous problems, where 2.1 is reduced to the Laplace equation. Each of these PDEs are subject to the appropriate boundary conditions on the domain boundary $\partial\Omega = \Gamma$. Such boundary conditions (BCs) can either be Dirichlet (Type-I), Neumann (Type-II), or Robin/Mixed (Type-III) BCs. Dirichlet problems impose the value of u on the boundaries, Neumann problems impose the flux or normal gradient $\partial_n u$ on the boundaries, while Robin problems impose a linear combination of Dirichlet and Neumann type BCs.

Although analytical solutions exist for some simple variations of the problems above, we are interested in looking at numerical methods to solve these equations. To do so, we look at various ways to discretize the domain Ω. This discretization will lead to a linear system of equations which we will solve with numerical methods, taking advantage of the sparsity and structure of the associated system.

2.2 Numerical Methods for Elliptic PDEs

2.2.1 Finite Difference

If Ω is on a logically rectangular domain (i.e. 2D Cartesian plane), perhaps the simplest discretization of the domain Ω is by collocating a mesh of points throughout the domain. Given upper and lower bounds in both directions, $[x_l, x_u], [y_l, y_u]$, the x- and y-location of each point can be defined as

$$x_i = x_l + i\Delta x, \quad i = 0, 1, ..., N_x - 1 \tag{2.3}$$

$$y_i = y_l + j\Delta y, \quad j = 0, 1, ..., N_y - 1 \tag{2.4}$$

where N_x, N_y is the number of points in the x- and y-direction and grid spacing is defined as

$$\Delta x = \frac{x_u - x_l}{N_x - 1} \tag{2.5}$$

$$\Delta y = \frac{y_u - y_l}{N_y - 1}. \tag{2.6}$$

These points are shown in the first plot of Figure 2.1. This type of discretization is known as the *finite difference method*. (Cite something here?). In the finite difference approach, the expressions for the derivatives in a PDE are replaced with Taylor series approximations. For example, a second order accurate, central-difference approximation to the second derivative can be given as

$$\frac{\partial^2 u}{\partial x^2} = \frac{u(x - \Delta x, y) - 2u(x, y) + u(x + \Delta x, y)}{\Delta x^2} + \mathcal{O}(\Delta x^2) \tag{2.7}$$

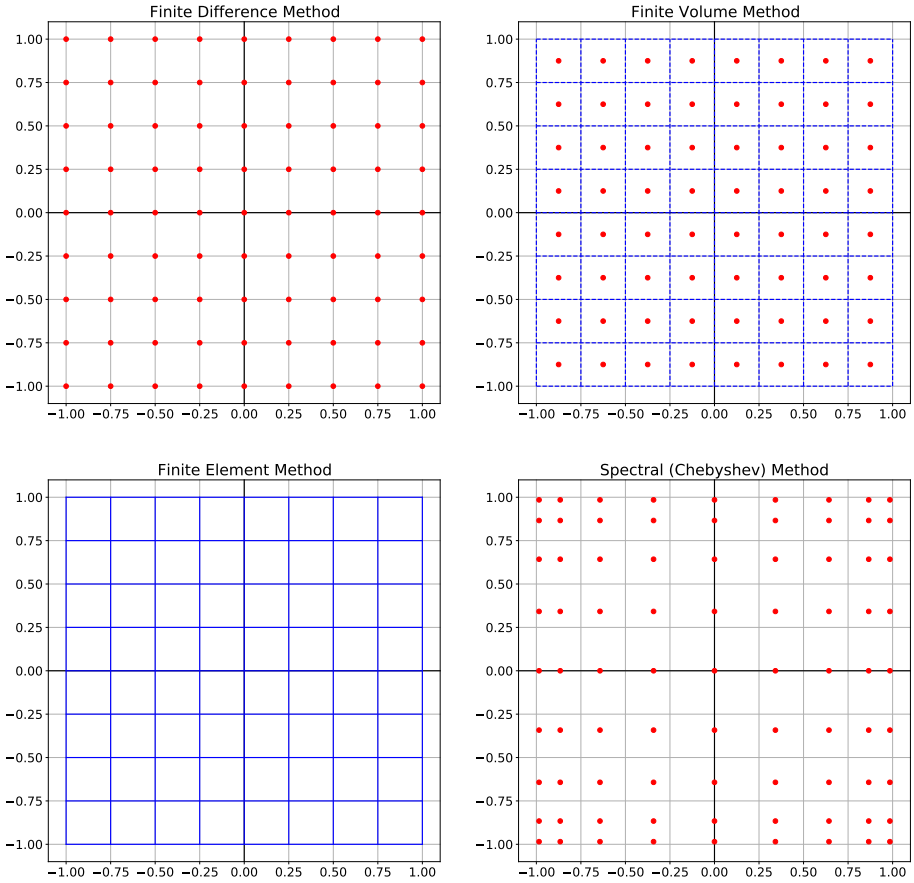


Figure 2.1: Discretization Methods. Top Left: Finite difference grid based on a mesh of points or nodes. Top Right: Finite volume mesh made of cells (dashed blue boxes) with cell averages in the center (red points). Bottom Left: Finite element mesh where each blue box is an element with a shape function defined at all points within the element. Bottom Right: Spectral mesh made up of a tensor of Chebyshev nodes.

and if we let $u(x_i, y_j) = u_{i,j}$, then we can write this as

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \mathcal{O}(\Delta x^2). \tag{2.8}$$

Using these Taylor Series expansions, we can replace the continuous Poisson's equation 2.1 with the discrete system of equations:

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} = f_{i,j}. \tag{2.9}$$

For each grid point $i = 0, ..., N_x - 1$, $j = 0, ..., N_y - 1$ this expression forms a linear system with $(N_x - 1) \times (N_y - 1)$ unknowns.

The system that is formed from this discretization is sparse and banded. For example, let $N_x = N_y = N$ for simplicity (thus $\Delta x = \Delta y = h$). Index the vector $\mathbf{u}$ first by rows, then by columns, such that

$$\mathbf{u} = [u_{0,0}, ..., u_{N-1,0}, u_{0,1}, ..., u_{N-1,1}, ..., u_{0,N-1}, ..., u_{N-1,N-1}]^T. \tag{2.10}$$

The corresponding linear system from 2.9 is the following:

$$\mathbf{A} = \frac{1}{h^2} \begin{bmatrix} \mathbf{D} & \mathbf{I}_N & \mathbf{0} & ... & \mathbf{0} \\ \mathbf{I}_N & \mathbf{D} & \mathbf{I}_N & ... & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_N & \mathbf{D} & ... & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & ... & \mathbf{D} \end{bmatrix} \tag{2.11}$$

where

$$\mathbf{D} = \begin{bmatrix} -4 & 1 & 0 & ... & 0 \\ 1 & -4 & 1 & ... & 0 \\ 0 & 1 & -4 & ... & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & ... & -4 \end{bmatrix}. \tag{2.12}$$

We also index the RHS similarly to $u(x, y)$ resulting in a RHS vector $\mathbf{f}$. This leads to the linear system

$$\mathbf{Au} = \mathbf{f} + \mathbf{b} \tag{2.13}$$

where $\mathbf{b}$ contains any updates necessary for any type of boundary condition.

2.2.2 Finite Volume

In a finite volume scheme, the domain is broken up into cells, which can be structured or unstructured polygons (2D) or polyhedrons (3D). The function to approximate is solved for in terms of a cell average. Finite volume schemes are often used to solve equations modeling conservation laws, where the cell quantity is conserved in time through balancing fluxes (what comes in and out of the cell boundaries) and the cell source (what the cell is generating or destroying). In a finite volume scheme, the cell average is updated according to the integral formulation of the conservation law as

$$\frac{\partial}{\partial t}\int_{\Omega_i} \mathbf{q} d\Omega_i + \int_{\Gamma_i} \mathbf{F}(\mathbf{q}) \cdot \hat{n}_i d\Gamma_i + \int_{\Omega_i} \mathbf{s} d\Omega_i = 0, \tag{2.14}$$

where Ω_i and Γ_i are the cell domains and boundaries, respectively, $\mathbf{q}$ is a vector of quantities in each cell, $\mathbf{F}$ is a flux function, and $\mathbf{s}$ is the cell's source term. In the second plot of Figure 2.1, the blue dashed lines are the cell boundaries, and the red points are the cell centers. For up to second order schemes, the cell average is collocated at the center of the cell; this changes for higher order schemes.

2.2.3 Finite Element

In a finite element scheme, the domain is broken into elements. These elements can also be structured or unstructured polygons (2D) or polyhedrons (3D), similar to the finite volume method. The PDE to solve is converted into a weak form by multiplying the PDE by a test function $v(x, y)$ and integrating over the domain:

$$\int_\Omega v\nabla^2 u d\Omega = \int_\Omega v f d\Omega. \tag{2.15}$$

Green's Theorem (i.e., integration by parts) is used to break the LHS into integrals on the domain Ω and the boundary $\partial\Omega = \Gamma$:

$$-\int_\Omega \nabla v \cdot \nabla u d\Omega + \int_\Gamma v\frac{\partial u}{\partial n} d\Gamma = \int_\Omega v f d\Omega \tag{2.16}$$

The idea behind the finite element method is to use basis functions

$$u(x, y) \approx \bar{u} = \sum_{k=1}^{N_k} c_k \phi(x, y) \tag{2.17}$$

inside each element Ω_i, where $\Omega = \cup_{i=1}^{N_{\text{elem}}} \Omega_i$. When the same basis functions for the test functions v are also used, this leads to the Galerkin finite element method. Upon substitution of the above into 2.16, and integrating the basis function with either

quadrature or with analytical expressions, a linear system is formed for the coefficients c_k.

There are several variations of the finite element method. The method explained above forms a global linear system involving all elements. To avoid this, mapping a physical element to some reference element, makes the contributions from a single element only influence itself and its neighbors. This makes the coefficient matrix in the linear system much more sparse and structured, making it easier to solve. Additionally, by not imposing continuity across element boundaries, one arrives at the discontinuous Galerkin method, where the discontinuities are handled through element fluxes. By choosing certain shape or basis functions, one can derive different variations of the finite element method, including the B-spline finite element method Kagan *et al.* (1998) and the spectral element method Patera (1984).

2.2.4 Spectral Methods

In spectral methods, the approach is to approximate the solution u as a linear combination of a finite set of orthogonal basis functions

$$u(x,y) = \sum_{i=1}^{N_x} \sum_{j=1}^{N_y} c_{i,j} \phi_i(x) \phi_j(y) \tag{2.18}$$

where the coefficients are chosen to minimize a norm, such as the L^2 norm of the residual $r(x,y) = \nabla^2 u(x,y) - f(x,y)$. On a discrete domain, this is similar to requiring $\nabla^2 u(x_i, y_j) = f(x_i, y_j)$ at all interior grid points. As this acts like an interpolation scheme, increasing the number of discretization points on with a fixed interval actually leads to highly oscillatory results. Thus, in spectral methods, it is common to use grid points that are clustered near the ends of the interval, such as Chebyshev points,

defined on an interval $[a, b]$ as

$$x_i = a + \frac{1}{2}(b-a)\Big(1 + \cos\big(\pi(1 - \frac{i}{N+1})\big)\Big), i = 0, ..., N+1. \tag{2.19}$$

These points are shown on the last plot of Figure 2.1. With a good basis of grid points, spectral methods can achieve very fast convergence. The linear system formed by this method will be dense as it functions like a high-order interpolation scheme. However, as convergence is much faster than high-order finite difference methods, one can use far fewer points on a grid, so the size of the system is kept small. If one uses Fourier series as the basis functions ϕ, one can accelerate the solution of the linear system using fast Fourier Transform algorithms. Though, due to the periodic nature of the Fourier series, it is more difficult to implement non-periodic boundary conditions (LeVeque (2007), Townsend & Olver (2015)).

2.2.5 Other Methods and Summary

These methods are not all the possible ways to solve an elliptic PDE. Other methods include using quadrature methods on the integral formulation of the PDE, as well as collocation methods such as radial basis function approximations. However, the methods we reviewed show some of the most classic approaches to solving elliptic PDEs.

2.3 Solution Methods for Elliptic Partial Differential Equations

The discretization methods described in Section 2.2 result in a linear system. To generally talk about these solution methods, we assume that we form the linear system

$$\mathbf{Au} = \mathbf{b} \tag{2.20}$$

where $\mathbf{A} \in \mathbb{R}^{N \times N}$ is a coefficient matrix formed from one of the discretization methods, N is the number of degrees of freedom, $\mathbf{u} = u_i = u(x_i)$ for $i \in \mathbf{I}_x$, index set $\mathbf{I}_x$ for the discrete domain, and $\mathbf{b}$ is the right-hand side vector encoded with the boundary conditions and the inhomogeneous function $f(\mathbf{x})$. The goal is to solve for $\mathbf{u}$ where we take advantage of the sparsity and structure of $\mathbf{A}$. We organize the various methods into the following three categories: 1) iterative methods, 2) direct methods, and 3) hierarchical methods.

2.3.1 Iterative Methods

Iterative methods start with an initial guess of a solution to 2.20 and correct the iterate until convergence to a specified tolerance. The simplest iterative methods are called splitting methods where the linear system is modified according to

$$\mathbf{A} = \mathbf{M} - \mathbf{N} \Rightarrow \mathbf{Mu} = \mathbf{Nu} + \mathbf{b}. \tag{2.21}$$

This suggests the following recursion relationship for the next iteration:

$$\mathbf{M}\mathbf{u}^{k+1} = \mathbf{N}\mathbf{u}^{k} + \mathbf{b} \tag{2.22}$$

$$\mathbf{u}^{k+1} = \mathbf{M}^{-1}\mathbf{N}\mathbf{u}^{k} + \mathbf{M}^{-1}\mathbf{b}. \tag{2.23}$$

The idea is to choose $\mathbf{M}$ that captures as much of $\mathbf{A}$ as possible, but is still easy and quick to invert. As $\mathbf{A}$ is either banded or sparse, classical splitting methods for elliptic PDEs split $\mathbf{A}$ into $\mathbf{A} = \mathbf{D} - \mathbf{L} - \mathbf{U}$, where $\mathbf{D}$ is the diagonal components of $\mathbf{A}$, and $\mathbf{L}$ and $\mathbf{U}$ are the lower and upper pieces, respectively. Classical choices for $\mathbf{M}$ and $\mathbf{N}$ are summarized in Table 2.1.

Jacobi	$\mathbf{M} = \mathbf{D}$	$\mathbf{N} = \mathbf{L} + \mathbf{U}$
Gauss-Sidel	$\mathbf{M} = \mathbf{D} - \mathbf{L}$	$\mathbf{N} = \mathbf{U}$
Successive Over Relaxation	$\mathbf{M} = \frac{1}{\omega}(\mathbf{D} - \omega\mathbf{L})$	$\mathbf{N} = \frac{1}{\omega}\big((1-\omega)\mathbf{D} + \omega\mathbf{U}\big)$

Table 2.1: Iterative Methods: Splitting Methods

Another class of iterative methods are called Krylov subspace methods. The Krylov space is defined as

$$\mathcal{K}_k = \text{span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, ..., \mathbf{A}^{k-1}\mathbf{r}_0\} \tag{2.24}$$

based on the initial residual $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{u}^{(0)}$. The goal is to take the next iteration from this particular space. Two common Krylov methods are the Conjugate Gradient method Hestenes *et al.* (1952) and the Generalized Minimal Residual (GMRES) method Saad & Schultz (1986). In the conjugate gradient method, the approximation is adjusted by a conjugate direction, or a vector that is conjugate with respect to $\mathbf{A}$. This vector is called the search direction $\mathbf{p}$ and is scaled by α, which is computed

by solving a quadratic minimization problem. The GMRES method builds up an orthogonal matrix $\mathbf{Q}$ through a process called the Arnoldi iteration. The Arnoldi iteration forms $\mathbf{A} = \mathbf{QHQ}^*$ for orthogonal matrix $\mathbf{Q}$ and Hessenberg matrix $\mathbf{H}$. The next iteration is found via $\mathbf{Qy}$ where $\mathbf{y}$ is found from a least squares problem involving $\mathbf{H}$. These methods are summarized in Table 2.2.

Conjugate Gradient	$\mathbf{u}^{(k+1)} = \mathbf{u}^{(k)} + \alpha^{(k)}\mathbf{p}^{(k)}$
GMRES	$\mathbf{u}^{(k)} = \mathbf{Q}^{(k)}\mathbf{y}$

Table 2.2: Iterative Methods: Krylov Subspace Methods

Most iterative methods are considered "matrix-free" methods. A "matrix-free" method is a method that does not explicitly form the matrix $\mathbf{A}$, but is rather applied to a vector. For example, if implementing a Conjugate Gradient method for a finite difference discretization, one has to compute the product $\mathbf{Au}^{(k)}$. Instead of doing the full matrix-vector calculation, one can write a function that takes $\mathbf{u}$ and returns the 2nd-order, central difference operator as computed in 2.9.

As finite difference discretization schemes for elliptic PDEs lead to sparse matrices, the application of $\mathbf{A}$ to a vector can be done in $\mathcal{O}(N)$ operations, where N is the number of unknowns in the vector. Thus, the performance for most iterative methods is approximately $\mathcal{O}(N \times N_{iter})$, where N_{iter} is the number of iterations required for a specified tolerance. However, as N gets larger, often so does N_{iter}, leading to poor scaling, as noted in Martinsson (2019). In addition, iterative methods may not always converge for a given initial guess or structure of $\mathbf{A}$, which make them unfavorable for "black-box" implementations for linear solvers.

2.3.2 Direct Methods

Motivation for direct solvers stems from wanting to improve upon the disadvantages of iterative methods. Martinsson notes in P. Martinsson (2004) some advantages to using direct methods over iterative ones:

- Direct methods can be applied to multiple right-hand side vectors $\mathbf{b}$ or multiple boundary conditions once a factorization or solution operator is built, whereas iterative methods must be solved anew for each right-hand side or for different boundary conditions.

- Direct methods can take advantage of "close" matrices (i.e. if we have an inverse or factorization of $\mathbf{A}$ and perturb it by ϵ, we could adjust the inverse to account for it instead of recompute the inverse).

- Most direct methods can take advantage of fast and efficient algorithms for matrix factorization such as the singular value decomposition, LU decomposition, QR decomposition, etc.

We will look at how direct methods are useful as we consider some common direct methods from LeVeque (2007) and Trefethen & Bau III (1997).

Many direct methods are based on matrix factorizations. Perhaps the most well-known is the LU decomposition. LU decomposition factors the coefficient matrix into a lower and upper triangular matrix: $\mathbf{A} = \mathbf{LU}$. The idea is to use Gaussian elimination to eliminate entries below the main diagonal, and then use back-substitution to solve for each entry in $\mathbf{u}$. In general, LU decomposition requires $\mathcal{O}(N^3)$ floating point operations and thus is impractical for large matrices. There are banded solvers for Gaussian elimination that can take advantage of the sparsity of a matrix. The Cholesky

decomposition is a variant of Gaussian elimination for symmetric matrices. Other matrix factorizations include the QR-decomposition, $\mathbf{A} = \mathbf{QR}$, and the singular value decomposition, $\mathbf{A} = \mathbf{U\Sigma V}^*$.

For a finite difference discretization of elliptic PDEs, $\mathbf{A}$ is banded and sparse, and more efficient algorithms for LU decomposition exist. For 1D problems, $\mathbf{A}$ is diagonally dominant and sparse with the bandwidth (the number of entries off the main diagonal in a matrix) dependent on the order of the stencil. For the stencil shown in the second order discretization in 2.9, $\mathbf{A}$ is tridiagonal. This allows us to use algorithms such as Thomas's algorithm for solving a tridiagonal system in $\mathcal{O}(N)$ steps (Higham (2002)). In higher dimensions, block versions of Thomas's algorithm exist (Quarteroni *et al.* (2010)).

Compared to iterative methods, direct methods will terminate in a finite number of steps. Direct methods are more fit for "black-box" implementations. However, because most direct methods need to explicitly form $\mathbf{A}$, they are more difficult to implement with limited computing resources. Indeed, going to higher dimensions and higher orders often dramatically increases memory and compute requirements.

2.3.3 Hierarchical Methods

The methods grouped under hierarchical methods attempt to accelerate some of the ideas from iterative and direct methods by breaking the problem into a hierarchy of subproblems. By recursively breaking the original problem into smaller subproblems, significant improvements can be made in complexity and performance. We'll talk about three hierarchical methods here: the multigrid method, nested dissection, and the Hierarchical Poincaré-Steklov (HPS) method.

The Multigrid Method

The multigrid method was introduced by Brandt in Brandt (1977). It has been widely used in various applications and for all types of solution methods. Briggs in Briggs *et al.* (2000) gives an overview and tutorial of the multigrid method.

In the multigrid method, the idea is to use multiple levels of grids and solution methods on each level to solve a larger problem. To look at the multigrid method, we define the error in the linear system to be $\mathbf{e} = \mathbf{u}^{(k)} - \mathbf{u}_{exact}$ (the difference between the exact solution and the kth iteration). After a few iterations of a smoother (typically Jacobi's method), because of the local averaging, any high-frequency error is quickly dampened away. What takes longer to eliminate is the low-frequency error associated with the global problem. By coarsening the grid, the lower frequency error is dampened quicker. Multigrid combines the ability of iterative methods to locally reduce error and a coarsening grid technique to accelerate convergence.

In the multigrid method, one starts with the solution on a fine grid, for example, with grid spacing h, and performs a few iterations of a smoother. After a few iterations, the h-level grid is projected onto a coarser $2h$-level grid. On this level, one performs a few more iterations of a smoother. Because the grid is coarser, this step is faster. Again, after a few iterations, the solution is projected onto an even coarser $4h$-level grid and a few more iterations are performed. This is done a specified number of times, and then the solution is interpolated back up the levels to the original grid. This is shown in Figure 2.2.

In what is called the "full multigrid" method, the process starts on the coarsest level instead of the finest. The solution is relaxed on this level, and then interpolated down onto a finer mesh. The interpolated solution is used as an initial guess for

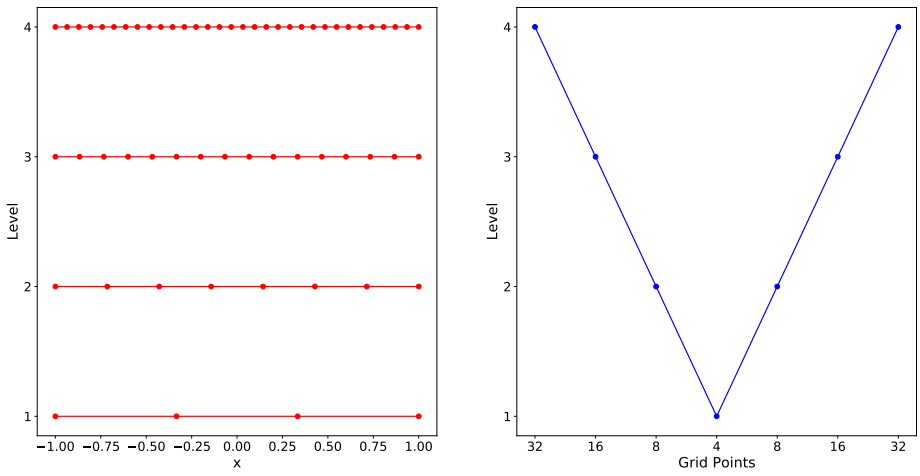


Figure 2.2: The 1D Multigrid Method. Starting with the finest grid, perform a few iterations of an iterative method. This is called relaxing the solution. Then project onto a coarser grid, and relax again. Do this down the levels in the grid to a desired precision. Once the solution is converged to on the coarsest level, interpolate back up the levels to obtain the solution on the finest level.

solving the problem on the finer mesh. The relaxed solution on a coarser grid is often an ideal initial guess for the problem on the finer mesh, resulting in quick convergence on that level.

The multigrid method allows one to accelerate an iterative solver. Multigrid methods are very effective as pre-conditioners for iterative methods. This is the general pattern of hierarchical methods: the ability to use classical iterative and direct methods on smaller grids where they perform well, and then "scale" them up to larger problem sizes.

Nested Dissection

The nested dissection method formulated by George in George (1973) is a direct method that builds upon Gaussian elimination for problems on a grid. It is also the

basis for forming what is called the multifrontal method. By taking advantage of the ordering of points on a grid, one can permute $\mathbf{A}$ to first eliminate points that split the mesh into two unconnected meshes. This permutation takes the form of $\mathbf{P}^*\mathbf{AP}$, where the goal is to form $\mathbf{P}$ to reorganize $\mathbf{A}$ in a way that eliminates points in an efficient manner.

To detail nested dissection better, consider an $N \times N$ mesh such as a finite difference mesh discussed in Section **??**. Assume that the initial ordering of points corresponds to an index set that iterates over the points in the mesh row-by-row. Now, the idea of nested dissection is to reorganize the points such that we first eliminate points down the middle of the mesh (i.e. the points at $x = 0$ in the first plot of Figure 2.1). Once these points are solved for, it spilts the mesh into two equally sized pieces that are disconnected. Each of the disconnected meshes now are smaller, and thus easier to solve. This idea of splitting the mesh into disconnected pieces by first eliminating points along an interface can be recursively applied to each split. This means that after dividing the mesh into two, one can divide those two meshes into four meshes, and so on. Recursive splitting is a common characteristic in hierarchical methods.

Nested dissection was first introduced by George in George (1973), and further generalized by Lipton et al. in Lipton *et al.* (1979). Martinsson has a tutorial on nested dissection in Martinsson (2019). In fact, nested dissection served as motivation for another hierarchical method proposed by Martinsson and Gillman called the Hierarchical Poincaré-Steklov method.

The Hierarchical Poincaré-Steklov Method

The work done by Gillman and Martinsson in P. Martinsson (2004), Martinsson (2013), and Gillman & Martinsson (2014) (with a practical tutorial found in Martinsson (2015)) culminate in what they call the Hierarchical Poincaré-Steklov (HPS) method. It is a direct solver for elliptic PDEs that is based on a binary tree of rectangular patches where the solution operator to $\mathbf{A}$ is built by recursively merging child patches. Like direct methods, the HPS method involves a factorization step and a solve step. The chief advantage of the HPS method over other direct methods is that it does not require the explicit formulation and storage of $\mathbf{A}$.

The HPS method starts with an original problem domain, and recursively divides the domain in half. This creates a binary tree of patches as shown in Figure 2.4. Once the domain has been decomposed into this tree of patches, two operators are defined on the lowest level, called the leaf level. These operators are the solution operator $\mathbf{S}$ and Dirichlet-to-Neumann (DtN) operator $\mathbf{T}$. The solution operator maps boundary data to solution data on the interior of the patch (i.e. solves the local boundary value problem), and the DtN operator maps Dirichlet data on the boundary to Neumann data on the boundary. These operators can be formed using any elliptic PDE solver, including fast solvers like spectral methods. After forming these operators, the next step is to recursively merge each sibling patch up the tree. the merge step is demonstrated in Figure 2.3. This results in a global solution operator that can be stored and used multiple times (at different time steps or with varying boundary conditions, etc.), and is similar to a direct method matrix factorization. The final step is applying the solution operator to each level down the tree to obtain the solution everywhere in the domain. This step is just a matrix-vector multiplication and is very

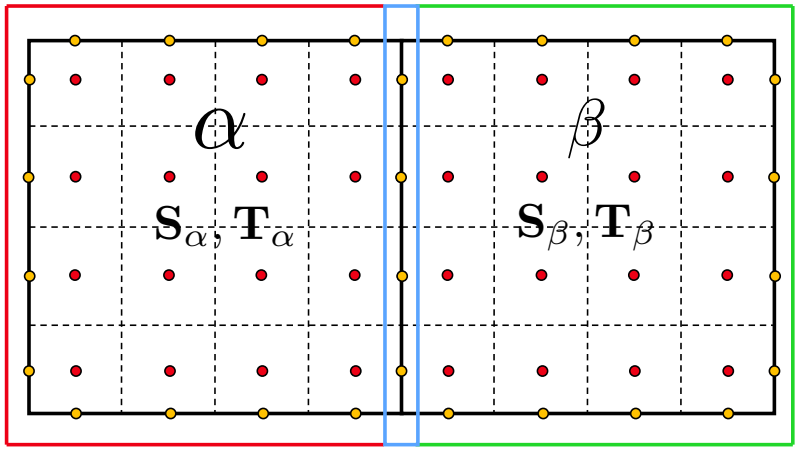


Figure 2.3: HPS Merge Operation. The merged patch Ω_τ is the union of children Ω_α and Ω_β, i.e. $\Omega_\tau = \Omega_\alpha \cup \Omega_\beta$. Red, green, and blue nodes correspond to index sets $\mathbf{I}_1$, $\mathbf{I}_2$, and $\mathbf{I}_3$, respectively. The merge operation eliminates the nodes on the interface of the children patches.

fast.

Similar to other direct methods, the HPS method forms an in-memory solution operator that can be applied to several right-hand side vectors. This property makes it ideal for problems where several elliptic solves are necessary. While most iterative methods have better asymptotic performance than other direct methods, the HPS method can be accelerated using hierarchically block seperable (HBS) matrix algebra to achieve near linear asymptotic performance (Gillman & Martinsson (2014)).

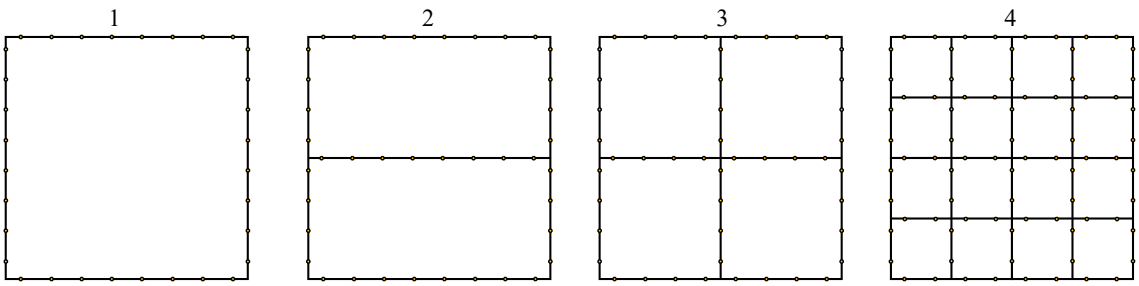


Figure 2.4: HPS Solve Stage. Once S_0 is formed, apply it to the top level Dirichlet data to get boundary (and solution) data on the interface of the children. Apply the patch solution operator down the tree until each leaf has it's local boundary information. Then apply the solution operator to get the solution data in the interior of each leaf.

CHAPTER 3:

THE QUADTREE-ADAPTIVE HPS METHOD

3.1 Introduction

Elliptic partial differential equations (PDEs) arise in fluid modeling, electromagnetism, astrophysics, heat transfer, and many other scientific and engineering applications. Solving elliptic PDEs numerically can place significant computational demands on scientific codes, and so development of fast, efficient solvers continues to be an active area of research. Efficient solvers based on finite difference or finite element methods will typically take advantage of the sparsity of the underlying linear systems and any mesh regularity. Efficient solvers have been developed (UMFPACK Davis (2004), FISHPACK Swarztrauber *et al.* (1999)) for uniform Cartesian meshes, since for these problems, the resulting linear system has a regular sparsity pattern that can be exploited.

For many applications, uniform Cartesian meshes are prohibitively expensive and so local mesh adaptivity can be used to resolve detailed solution features. One common approach to Cartesian mesh adaptivity is the patch-based method, originally developed by Berger, Oligier and later Colella. Software libraries implementing methods related to this patch-based approach include Chombo Colella *et al.* (2009), AMReX Zhang *et al.* (2019), and SAMRAI Hornung *et al.* (2006). In a patch-based approach, the

computational mesh is defined as a union of overlapping rectangular patches, with finer patches placed on top of coarser patches in regions where resolution for detailed solution features is needed. These rectangular patches are of arbitrary size, but align with mesh coordinates of the underlying coarser mesh. Proper nesting rules ensure that every patch is completely contained within coarser patches at the same coarse resolution.

A second approach to Cartesian mesh adaptivity, and the approach used in this paper, is to construct a composite mesh by filling leaves of an adaptive quadtree with non-overlapping locally Cartesian grids of the same size. Quadrants in the quadtree are refined by subdividing the quadrant into four smaller quadrants, each occupying one quarter of the space of the larger quadrant. Similarly, coarsening occurs when four finer quadrants are replaced by a single coarser quadrant. Every quadrant in the quadtree layout contains a mesh of the same size, but since each quadrant occupies a space determined from their level in the quadtree, the effective resolution of each grid is finer or coarser, depending on their level in the quadtree. Adaptive, Cartesian software libraries using a tree-based approach include PARAMESH Globisch (1995), FLASH-X Dubey *et al.* (2022) and ForestClaw Calhoun & Burstedde (2017).

Solving elliptic problems on an adaptive hierarchy of meshes introduces technical challenges that are not present with uniform Cartesian meshes. Methods that require matrix assembly are much more difficult to use on adaptive meshes, since row entries corresponding to discretizations at boundaries between coarse and fine meshes are based on non-trivial stencils. The Hypre library Falgout & Yang (2002) provides some tools for matrix assembly, but these tools are not immediately useful in adaptive mesh case. A more common approach is to use matrix-free methods such as multigrid

or Krylov methods. Multigrid may be particularly well suited for adaptive meshes, since the adaptive levels are automatically built into the mesh hierarchy. AMReX, for example, makes extensive use of multigrid for their solvers Zhang *et al.* (2019). However, the performance of iterative solvers is largely problem dependent and also affected by irregular stencils at boundaries between coarse and fine meshes.

In Gillman & Martinsson (2014), Gillman and Martinsson describe a direct solver for elliptic problems that is particularly well suited to composite quadtree meshes. In their approach, a matrix-free factorization of the linear system is constructed by merging leaf level quadrants in a uniformly refined composite quadtree mesh up to a root node of the tree. Then in a solve stage, the boundary condition data imposed on the physical domain is propagated back down the tree, where Dirichlet problems on each leaf patch can be solved using fast solvers. This approach does not require any matrix assembly on the composite mesh and can be used with any uniform grid solver at the leaf level. In their original work, Gillman and Martinsson use high-order spectral collocation methods and low-rank optimization to achieve $\mathcal{O}(N)$ factorization complexity Gillman & Martinsson (2014). This method has come to be known as the Hierarchical Poincaré-Steklov (HPS) method Martinsson (2015). Advancements to the HPS method include 3D and parallel implementations Hao & Martinsson (2016); Beams *et al.* (2020). Applications of the HPS method can be found in Fortunato *et al.* (2020). More details on the HPS method can be found in Chapters 19–27 of Martinsson (2019) and a tutorial on the HPS method can be found in Martinsson (2015).

The focus of this paper is on an implementation of the HPS method on an adaptive, finite volume mesh using the state-of-the-art meshing library `p4est` to generate the

adaptive quadtree mesh Burstedde *et al.* (2011). The `p4est` library provides highly efficient data structures and iterators necessary for implementing the HPS method. A feature that sets `p4est` apart from other quadtree meshing libraries is that it multiblock meshes (e.g. a "forest-of-octrees"), allowing for significant flexibility in the geometry of the mesh. `p4est` uses a space filling curve to partition the mesh using MPI, and provides utilities for dynamic adaptivity. Our work follows upon earlier work presented in Babb *et al.* (2018); Geldermans & Gillman (2019).

3.1.1 Problem Statement

We focus on the constant coefficient elliptic problem

$$\nabla^2 u(x,y) + \lambda u(x,y) = f(x,y), \quad \lambda \geq 0 \tag{3.1}$$

with $(x,y) \in \Omega = [a,b] \times [c,d]$, subject to Dirichlet boundary conditions

$$u(x,y) = g(x,y), \quad (x,y) \in \Gamma = \partial\Omega. \tag{3.2}$$

For this paper, we assume that $b - a = d - c$ so that we can describe our mesh using a single quadtree and square mesh cells. In general though, this is not a limitation of meshes generated using the multi-block "forest-of-octrees" capabilities of `p4est`. The primary contributions of this paper are an HPS solver for the constant coefficient ellitpic problem on an adaptively refined `p4est` mesh. In future work, we will include fast linear algebra needed to reduce the complexity of the overall algorithm to $\mathcal{O}(N)$, a key contribution of the original HPS method developed by Gillman and Martinsson.

In Section 3.2, we describe the quadtree-adaptive HPS method, including the leaf-level computations, merging and splitting algorithms, and the handling of coarse-

fine interfaces. This section also includes a comparison between our 4-to-1 approach contrasted to the 2-to-1 approach presented in Gillman & Martinsson (2014). Section 3.3 describes implementation details such as the data structures and functions that wrap the `p4est` mesh backend. Finally, in Section 3.4 we provide results of numerical experiments solving different types of elliptic PDEs, including a convergence analysis and timing results for the use of the quadtree-adaptive HPS method on adaptive meshes.

3.2 Components of the HPS algorithm : Merging, Splitting and Leaf Computations

The HPS method applied to 3.1 includes a factorization stage and a solve stage. For problems in which for $f(x, y)$ is non-zero, an additional "upwards" stage is required for a particular right hand side. In our algorithm, the build and upwards stages are a successive application of a 4-to-1 merge algorithm while the solve stage is a an application of a 1-to-4 split algorithm. In this section, we describe these merging and splitting operations.

3.2.1 Leaf Level Computations

In what follows, we assume that each leaf-level quadrant in the quadtree mesh stores a uniform Cartesian grid with $M \times M$ mesh cells. We refer to this local Cartesian mesh, along with its quadrant as a *patch*. In Figure 3.1 (left), we show four patches in a larger composite domain. Cell-centered finite difference schemes are particularly convenient for adaptively refined Cartesian meshes, since the cell-centered values are not duplicated on adjacent patches.

At the leaf-level, the build stage of the HPS methods requires the solution to a

Poisson problem (3.1) discretized on an $M \times M$ cell-centered grid as

$$\mathcal{A}^\tau \mathbf{u}^\tau = \mathcal{B}^\tau \mathbf{g}^\tau + \mathbf{f}^\tau \tag{3.3}$$

where $\mathcal{B}^\tau$ spreads Dirichlet boundary data $\mathbf{g}^\tau \in \mathbb{R}^{4M}$ to the grid and $\mathbf{f}^\tau$ is the right hand side function $f(x, y)$ from (3.1) evaluated at cell centers. We write the solution to this problem in terms of a *homogeneous operator* $\mathcal{L}_h^\tau$ and an inhomogeneous operator $\mathcal{L}_{inh}^\tau$ which represent efficient solvers on the uniform Cartesian patch. The solution is then given by

$$\mathbf{u}^\tau = \mathcal{L}_h^\tau \mathbf{g}^\tau + \mathcal{L}_{inh}^\tau \mathbf{f}^\tau \tag{3.4}$$

Where the context is clear, we also view operators $\mathcal{L}_h^\tau$ and $\mathcal{L}_{inh}^\tau$, boundary data $\mathbf{g}^\tau$ and right hand side data $\mathbf{f}^\tau$ as matrices and vectors of the appropriate sizes. The components of $\mathbf{u}^\tau$ are given by u_{ij}^τ, for $i, j = 1, 2, \ldots M$.

Using these leaf-level solvers, we can now build a leaf-level DtN operator $\mathbf{T}^\tau$. In the description below, we illustrate the HPS method on the complete Poisson problem, assuming a non-zero right hand side. We remark below, though, that if one anticipates solving for multiple right-hand sides, the build stage can be separated into a factorization stage and an "upwards" stage to handle inhomogeneous data.

Dirichlet-to-Neumann (DtN) operator

Given a cell-centered grid solution $\mathbf{u}^\tau = \mathcal{L}_h^\tau \mathbf{g}^\tau + \mathcal{L}_{inh}^\tau \mathbf{f}^\tau$, we define "interior" grid values $\mathbf{u}^{(in)} \in \mathbb{R}^{4M}$ as those grid solution values $\mathbf{u}^\tau$ for which $i = 1$, $i = M$, $j = 1$ or $j = M$. By analogy, components of $\mathbf{u}^{(out)}$ are grid solution values for which $i = 0$ or $i = M + 1$ or $j = 0$ or $j = M + 1$.

On a cell-centered finite volume grid, the known Dirichlet boundary values are not collocated with components of the grid solution so we discretize the Dirichlet boundary condition at the midpoint between $u_k^{(in)}$ and $u_k^{(out)}$ as

$$g_k^\tau = \frac{u_k^{(out)} + u_k^{(in)}}{2}, \quad k = 1, \ldots, 4M \tag{3.5}$$

where g_k, $u_k^{(in)}$ and $u_k^{(out)}$ are the components of $\mathbf{g}^\tau$, $\mathbf{u}^{(in)}$ and $\mathbf{u}^{(out)}$ respectively.

By analogy, we discretize Neumann data on the boundary of the patch as

$$v_k = \frac{u_k^{(out)} - u_k^{(in)}}{h} \tag{3.6}$$

where h is the mesh cell width (assumed to be uniform in both x- and y- directions).

Eliminating $u_k^{(out)}$ between the two expressions, we can express the Neumann data v_k in terms of Dirichlet data g_k and the interior value $u_k^{(in)}$ as

$$v_k = \frac{2}{h}(g_k - u_k^{(in)}). \tag{3.7}$$

The vector of Neumann data on patch τ can then be expressed as

$$\mathbf{v}^\tau = \frac{2}{h}(\mathbf{g}^\tau - \mathbf{u}^{(in)}). \tag{3.8}$$

Introducing an operator (or a matrix of the appropriate size) $\mathbf{G}$ that selects $\mathbf{u}^{(in)}$ from the grid solution data $\mathbf{u}^\tau$, we write $\mathbf{u}^{(in)} = \mathbf{G}\mathbf{u}^\tau$ so that

$$\mathbf{v}^\tau = \frac{2}{h}(\mathbf{g}^\tau - \mathbf{G}(\mathcal{L}_h^\tau \mathbf{g}^\tau + \mathcal{L}_{inh}^\tau \mathbf{f}^\tau) = \frac{2}{h}(I - \mathbf{G}\mathcal{L}_h^\tau)\mathbf{g}^\tau - \frac{h}{2}\mathbf{G}\mathcal{L}_{inh}^\tau \mathbf{f}^\tau \tag{3.9}$$

From this, we define the leaf level Dirichlet-to-Neumann mapping $\mathbf{T}^\tau \in \mathbb{R}^{4M\times 4M}$ as

$$\mathbf{T}^\tau \equiv \frac{2}{h}(I - \mathbf{G}\mathcal{L}_h^\tau). \tag{3.10}$$

with inhomogenenous component

$$\mathbf{h}^\tau \equiv -\frac{2}{h}\mathbf{G}\mathcal{L}_{inh}^\tau \mathbf{f}^\tau. \tag{3.11}$$

Neumann data on the patch is then computed as

$$\mathbf{v}^\tau = \mathbf{T}^\tau \mathbf{g}^\tau + \mathbf{h}^\tau. \tag{3.12}$$

We refer $\mathbf{T}^\tau$ as a discrete Dirichlet-to-Neumann operator, although the Neumann data $\mathbf{v}^\tau$ is not the Neumann data that is commonly understood to be the result of applying a DtN map, since $\mathbf{v}^\tau$ depends not only on the Dirichlet data $\mathbf{g}^\tau$, but also on the inhomogeneous data $\mathbf{f}^\tau$.

3.2.2 The 4-to-1 Merge Algorithm

If τ is not a leaf-level quadrant, then we construct the DtN map $\mathbf{T}^\tau$ *recursively* by merging mappings from the four children patches α, β, γ and ω partitioning τ. For the non-leaf, two additional operators $\mathbf{S}^\tau$ and $\mathbf{X}^\tau$ (for non-zero $f(x, y)$) are also constructed. Prior to this merge process, it is assumed that each patch has computed a DtN mapping $\mathbf{T}^i$, for $i = \alpha$, β, γ, ω.

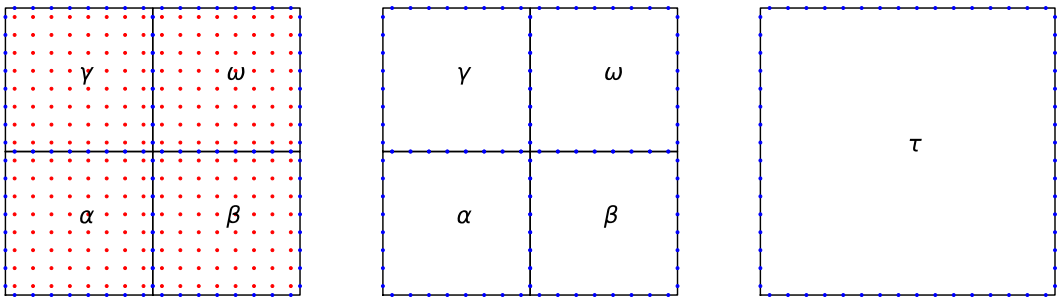


Figure 3.1: The 4-to-1 merge process: (left) the four children patches with their local grid, (middle) the four children share internal and external boundaries with τ, (right) the merged parent patch with data on the exterior of τ.

Merging DtN mappings

We partition sibling operators $\mathbf{T}^i$, $i = \alpha, \beta, \gamma, \omega$ according to how Dirichlet data on an edge of patch i are mapped to an edge on i containing Neumann data. For example, the sub-matrix $\mathbf{T}^{\alpha}_{\tau\gamma}$ maps Dirichlet data on the boundary shared between sibling patch α and parent patch τ to Neumann data on the boundary shared between patch α and sibling patch γ. In a similar manner, the Dirichlet data on each of the sibling patches i into $\mathbf{g}^{\alpha}_{\tau}$, $\mathbf{g}^{\alpha}_{\gamma}$ and $\mathbf{g}^{\alpha}_{\beta}$, where the subscripts indicate data on edges that patch α shares with patches τ, γ and β. Neumann data $\mathbf{v}^{\alpha}$ and $\mathbf{h}^{\alpha}$ are partitioned in an analogous manner.

With these partitions, we can write a set of equations for each of the four sibling

DtN mappings as

$$\mathbf{v}^{\alpha} = \mathbf{T}^{\alpha}\mathbf{g}^{\alpha} + \mathbf{h}^{\alpha} \quad \Longrightarrow \quad \begin{bmatrix} \mathbf{v}^{\alpha}_{\tau} \\ \mathbf{v}^{\alpha}_{\gamma} \\ \mathbf{v}^{\alpha}_{\beta} \end{bmatrix} = \begin{bmatrix} \mathbf{T}^{\alpha}_{\tau\tau} & \mathbf{T}^{\alpha}_{\gamma\tau} & \mathbf{T}^{\alpha}_{\beta\tau} \\ \mathbf{T}^{\alpha}_{\tau\gamma} & \mathbf{T}^{\alpha}_{\gamma\gamma} & \mathbf{T}^{\alpha}_{\beta\gamma} \\ \mathbf{T}^{\alpha}_{\tau\beta} & \mathbf{T}^{\alpha}_{\gamma\beta} & \mathbf{T}^{\alpha}_{\beta\beta} \end{bmatrix} \begin{bmatrix} \mathbf{g}^{\alpha}_{\tau} \\ \mathbf{g}^{\alpha}_{\gamma} \\ \mathbf{g}^{\alpha}_{\beta} \end{bmatrix} + \begin{bmatrix} \mathbf{h}^{\alpha}_{\tau} \\ \mathbf{h}^{\alpha}_{\gamma} \\ \mathbf{h}^{\alpha}_{\beta} \end{bmatrix}$$

$$\mathbf{v}^{\beta} = \mathbf{T}^{\beta}\mathbf{g}^{\beta} + \mathbf{h}^{\beta} \quad \Longrightarrow \quad \begin{bmatrix} \mathbf{v}^{\beta}_{\tau} \\ \mathbf{v}^{\beta}_{\omega} \\ \mathbf{v}^{\beta}_{\alpha} \end{bmatrix} = \begin{bmatrix} \mathbf{T}^{\beta}_{\tau\tau} & \mathbf{T}^{\beta}_{\omega\tau} & \mathbf{T}^{\beta}_{\alpha\tau} \\ \mathbf{T}^{\beta}_{\tau\omega} & \mathbf{T}^{\beta}_{\omega\omega} & \mathbf{T}^{\beta}_{\alpha\omega} \\ \mathbf{T}^{\beta}_{\tau\alpha} & \mathbf{T}^{\beta}_{\omega\alpha} & \mathbf{T}^{\beta}_{\alpha\alpha} \end{bmatrix} \begin{bmatrix} \mathbf{g}^{\beta}_{\tau} \\ \mathbf{g}^{\beta}_{\omega} \\ \mathbf{g}^{\beta}_{\alpha} \end{bmatrix} + \begin{bmatrix} \mathbf{h}^{\beta}_{\tau} \\ \mathbf{h}^{\beta}_{\omega} \\ \mathbf{h}^{\beta}_{\alpha} \end{bmatrix}$$

$$\mathbf{v}^{\gamma} = \mathbf{T}^{\gamma}\mathbf{g}^{\gamma} + \mathbf{h}^{\gamma} \quad \Longrightarrow \quad \begin{bmatrix} \mathbf{v}^{\gamma}_{\tau} \\ \mathbf{v}^{\gamma}_{\alpha} \\ \mathbf{v}^{\gamma}_{\omega} \end{bmatrix} = \begin{bmatrix} \mathbf{T}^{\gamma}_{\tau\tau} & \mathbf{T}^{\gamma}_{\alpha\tau} & \mathbf{T}^{\gamma}_{\omega\tau} \\ \mathbf{T}^{\gamma}_{\tau\alpha} & \mathbf{T}^{\gamma}_{\alpha\alpha} & \mathbf{T}^{\gamma}_{\omega\alpha} \\ \mathbf{T}^{\gamma}_{\tau\omega} & \mathbf{T}^{\gamma}_{\alpha\omega} & \mathbf{T}^{\gamma}_{\omega\omega} \end{bmatrix} \begin{bmatrix} \mathbf{g}^{\gamma}_{\tau} \\ \mathbf{g}^{\gamma}_{\alpha} \\ \mathbf{g}^{\gamma}_{\omega} \end{bmatrix} + \begin{bmatrix} \mathbf{h}^{\gamma}_{\tau} \\ \mathbf{h}^{\gamma}_{\alpha} \\ \mathbf{h}^{\gamma}_{\omega} \end{bmatrix}$$

$$\mathbf{v}^{\omega} = \mathbf{T}^{\omega}\mathbf{g}^{\omega} + \mathbf{h}^{\omega} \quad \Longrightarrow \quad \begin{bmatrix} \mathbf{v}^{\omega}_{\tau} \\ \mathbf{v}^{\omega}_{\beta} \\ \mathbf{v}^{\omega}_{\gamma} \end{bmatrix} = \begin{bmatrix} \mathbf{T}^{\omega}_{\tau\tau} & \mathbf{T}^{\omega}_{\beta\tau} & \mathbf{T}^{\omega}_{\gamma\tau} \\ \mathbf{T}^{\omega}_{\tau\beta} & \mathbf{T}^{\omega}_{\beta\beta} & \mathbf{T}^{\omega}_{\gamma\beta} \\ \mathbf{T}^{\omega}_{\tau\gamma} & \mathbf{T}^{\omega}_{\beta\gamma} & \mathbf{T}^{\omega}_{\gamma\gamma} \end{bmatrix} \begin{bmatrix} \mathbf{g}^{\omega}_{\tau} \\ \mathbf{g}^{\omega}_{\beta} \\ \mathbf{g}^{\omega}_{\gamma} \end{bmatrix} + \begin{bmatrix} \mathbf{g}^{\omega}_{\tau} \\ \mathbf{g}^{\omega}_{\beta} \\ \mathbf{g}^{\omega}_{\gamma} \end{bmatrix} \qquad . \tag{3.13}$$

Taking the first equation from each of the four sets of equations in (3.13), we write a system of equations for the Dirichlet data to get

$$A\mathbf{g}_{ext} + B\mathbf{g}_{int} + \mathbf{h}_{ext} = \mathbf{v}_{ext} \tag{3.14}$$

where the Dirichlet data has been partitioned into *exterior* data and *interior* data.

The exterior data is data on the boundary of the parent patch τ is given by

$$\mathbf{g}_{ext} \equiv \left[\mathbf{g}_\tau^\alpha, \mathbf{g}_\tau^\beta, \mathbf{g}_\tau^\gamma, \mathbf{g}_\tau^\omega\right]^T \tag{3.15}$$

To define the interior data, we use the fact that our solution is continuous across boundaries shared by sibling grids and enumerate data on the four shared boundaries as

$$\begin{aligned} \mathbf{g}_0 &\equiv \mathbf{g}_\gamma^\alpha = \mathbf{g}_\alpha^\gamma \\ \mathbf{g}_1 &\equiv \mathbf{g}_\omega^\beta = \mathbf{g}_\beta^\omega \\ \mathbf{g}_2 &\equiv \mathbf{g}_\beta^\alpha = \mathbf{g}_\alpha^\beta \\ \mathbf{g}_3 &\equiv \mathbf{g}_\omega^\gamma = \mathbf{g}_\gamma^\omega. \end{aligned} \tag{3.16}$$

With the Dirichlet data on the shared boundary uniquely defined, the interior partition of the Dirichlet data is given as

$$\mathbf{g}_{int} \equiv \left[\mathbf{g}_0, \mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3\right]^T \tag{3.17}$$

The Neumann data $\mathbf{v}^\tau$ and $\mathbf{h}^\tau$ is partitioned analogously.

The block matrices A and B are defined as

$$A = \begin{bmatrix} \mathbf{T}_{\tau\tau}^\alpha & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{T}_{\tau\tau}^\beta & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{T}_{\tau\tau}^\gamma & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{T}_{\tau\tau}^\omega \end{bmatrix}, \qquad B = \begin{bmatrix} \mathbf{T}_{\gamma\tau}^\alpha & \mathbf{0} & \mathbf{T}_{\beta\tau}^\alpha & \mathbf{0} \\ \mathbf{0} & \mathbf{T}_{\omega\tau}^\beta & \mathbf{T}_{\alpha\tau}^\beta & \mathbf{0} \\ \mathbf{T}_{\alpha\tau}^\gamma & \mathbf{0} & \mathbf{0} & \mathbf{T}_{\omega\tau}^\gamma \\ \mathbf{0} & \mathbf{T}_{\beta\tau}^\omega & \mathbf{0} & \mathbf{T}_{\gamma\tau}^\omega \end{bmatrix} \tag{3.18}$$

To obtain a second set of equations, we impose a continuity condition on the

normal derivative across edges shared between sibling patches and get

$$\begin{aligned}
\mathbf{v}_{\gamma}^{\alpha} + \mathbf{v}_{\alpha}^{\gamma} &= \mathbf{0} \\
\mathbf{v}_{\omega}^{\beta} + \mathbf{v}_{\beta}^{\omega} &= \mathbf{0} \\
\mathbf{v}_{\beta}^{\alpha} + \mathbf{v}_{\alpha}^{\beta} &= \mathbf{0} \\
\mathbf{v}_{\omega}^{\gamma} + \mathbf{v}_{\gamma}^{\omega} &= \mathbf{0}.
\end{aligned} \tag{3.19}$$

We organize the resulting four equations as

$$C\mathbf{g}_{ext} + D\mathbf{g}_{int} + \Delta\mathbf{h} = \mathbf{0} \tag{3.20}$$

where

$$C = \begin{bmatrix}
\mathbf{T}_{\tau\gamma}^{\alpha} & \mathbf{0} & \mathbf{T}_{\tau\alpha}^{\gamma} & \mathbf{0} \\
\mathbf{0} & \mathbf{T}_{\tau\omega}^{\beta} & \mathbf{0} & \mathbf{T}_{\tau\beta}^{\omega} \\
\mathbf{T}_{\tau\beta}^{\alpha} & \mathbf{T}_{\tau\alpha}^{\beta} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{T}_{\tau\omega}^{\gamma} & \mathbf{T}_{\tau\gamma}^{\omega},
\end{bmatrix} \tag{3.21}$$

$$D = \begin{bmatrix}
\mathbf{T}_{\gamma\gamma}^{\alpha} + \mathbf{T}_{\alpha\alpha}^{\gamma} & \mathbf{0} & \mathbf{T}_{\beta\gamma}^{\alpha} & \mathbf{T}_{\omega\alpha}^{\gamma} \\
\mathbf{0} & \mathbf{T}_{\omega\omega}^{\beta} + \mathbf{T}_{\beta\beta}^{\omega} & \mathbf{T}_{\alpha\omega}^{\beta} & \mathbf{T}_{\gamma\beta}^{\omega} \\
\mathbf{T}_{\gamma\beta}^{\alpha} & \mathbf{T}_{\omega\alpha}^{\beta} & \mathbf{T}_{\beta\beta}^{\alpha} + \mathbf{T}_{\alpha\alpha}^{\beta} & \mathbf{0} \\
\mathbf{T}_{\alpha\omega}^{\gamma} & \mathbf{T}_{\beta\gamma}^{\omega} & \mathbf{0} & \mathbf{T}_{\omega\omega}^{\gamma} + \mathbf{T}_{\gamma\gamma}^{\omega}
\end{bmatrix}. \tag{3.22}$$

and

$$\Delta\mathbf{h} = \begin{bmatrix}
\mathbf{h}_{\gamma}^{\alpha} + \mathbf{h}_{\alpha}^{\gamma} \\
\mathbf{h}_{\omega}^{\beta} + \mathbf{h}_{\beta}^{\omega} \\
\mathbf{h}_{\beta}^{\alpha} + \mathbf{h}_{\alpha}^{\beta} \\
\mathbf{h}_{\omega}^{\gamma} + \mathbf{h}_{\gamma}^{\omega}
\end{bmatrix}. \tag{3.23}$$

Combining (3.14) and (3.20), we have

$$\begin{aligned} A\mathbf{g}_{ext} + B\mathbf{g}_{int} + \mathbf{h}_{ext} &= \mathbf{v}_{ext} \\ C\mathbf{g}_{ext} + D\mathbf{g}_{int} + \Delta\mathbf{h} &= \mathbf{0} \end{aligned} \tag{3.24}$$

Writing this system as an augmented system and applying a single step of block Gaussian elimination, we get

$$\left[\begin{array}{cc|c} A & B & \mathbf{v}_{ext} - \mathbf{h}_{ext} \\ C & D & -\Delta\mathbf{h} \end{array}\right] \Longrightarrow \left[\begin{array}{cc|c} A - BD^{-1}C & 0 & \mathbf{v}_{ext} - \mathbf{h}_{ext} + BD^{-1}\Delta\mathbf{h} \\ D^{-1}C & I & -D^{-1}\Delta\mathbf{h} \end{array}\right] \tag{3.25}$$

We express the first row of the reduced system in equation form as

$$\left(A - BD^{-1}C\right)\mathbf{g}_{ext} = \mathbf{v}_{ext} - \mathbf{h}_{ext} + BD^{-1}\Delta\mathbf{h} \tag{3.26}$$

From this, we can identify the *merged* DtN operator

$$\mathbf{T}^{\tau} \equiv \mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C} \tag{3.27}$$

as the operator that maps exterior Dirichlet data to exterior Neumann data, with merged inhomogeneous Neumann data given as

$$\mathbf{h}^{\tau} \equiv -\mathbf{h}_{ext} + BD^{-1}\Delta\mathbf{h}. \tag{3.28}$$

With these choices of $\mathbf{T}^{\tau}$ and $\mathbf{h}^{\tau}$ we recover the expression from (3.12).

The second row of the reduced system (3.25), given by

$$\mathbf{g}_{int} = -D^{-1}C\mathbf{g}_{ext} - D^{-1}\Delta\mathbf{h}, \tag{3.29}$$

shows us how to recover interior Dirichlet data $\mathbf{g}_{int}$ from exterior data $\mathbf{g}_{ext}$. From this, we introduce the *solve* operator $\mathbf{S}^\tau$, given by

$$\mathbf{S}^\tau \equiv -D^{-1}C \tag{3.30}$$

Introducing the operator

$$\mathbf{X}^\tau \equiv -D^{-1} \tag{3.31}$$

we can write $\mathbf{T}^\tau$ and $\mathbf{S}^\tau$ in convenient form as

$$\begin{aligned} \mathbf{S}^\tau &= \mathbf{X}^\tau C \\ \mathbf{T}^\tau &= A + B\mathbf{S}^\tau \end{aligned} \tag{3.32}$$

To compute the merged inhomogeneous term, we first compute

$$\mathbf{w}^\tau \equiv \mathbf{X}^\tau \Delta\mathbf{h} \tag{3.33}$$

so that

$$\mathbf{h}^\tau = -\mathbf{h}_{ext} - B\mathbf{w}^\tau \tag{3.34}$$

Once the build stage is complete, the solve state maps Dirichlet data on parent

quadrants to Dirichlet data on child quadrants via the solve operator $\mathbf{S}^\tau$, as

$$\mathbf{g}_{int} = \mathbf{S}^\tau \mathbf{g}_{ext} + \mathbf{w}^\tau. \tag{3.35}$$

The formalism presented in terms of merged components $\mathbf{X}^\tau$, $\mathbf{S}^\tau$, $\mathbf{T}^\tau$ and $\mathbf{h}^\tau$ is identical to that presented in Martinsson (2015) for the horizontal merge of two leaf boxes.

The build stage of HPS method is described in Algorithm 1

Algorithm 1 Build stage on a uniformly refined quadtree mesh

for $\tau = 0, 1, \ldots$ **do** ▷ *Iterate over quadrants in uniform quadtree*

  if τ is a leaf-level patch **then**

    Build and store leaf-level DtN map $\mathbf{T}^\tau$ and inhomogeneous data $\mathbf{h}^\tau$

  else ▷ *τ has four child patches i, $i = \alpha, \beta, \gamma, \omega$.*

    Solve $D\mathbf{S}^\tau = -C$ to get operator $\mathbf{S}^\tau$.

    Build operator $\mathbf{T}^\tau$ using (3.32).

    Solve $D\mathbf{w}^\tau = \Delta\mathbf{h}$ using to get $\mathbf{w}^\tau$.

    Build merged inhomogeneous vector $\mathbf{h}^\tau$ using (3.34).

    Store $\mathbf{S}^\tau$, $\mathbf{T}^\tau$, $\mathbf{h}^\tau$ and $\mathbf{w}^\tau$ with quadrant τ.

    Delete operators $\mathbf{T}^\alpha$, $\mathbf{T}^\beta$, $\mathbf{T}^\gamma$, $\mathbf{T}^\omega$ and inhomogeneous data $\mathbf{h}^\alpha$, $\mathbf{h}^\beta$, $\mathbf{h}^\gamma$ and $\mathbf{h}^\omega$.

At the end of this build stage, we will have a single DtN operator $\mathbf{T}^\tau$ and single vector $\mathbf{h}^\tau$ for the root patch. At all levels, though, we will have a solve operator $\mathbf{S}^\tau$ and inhomogeneous vector $\mathbf{w}^\tau$.

A solve stage of the HPS method starts with data $g(x, y)$ defined on the boundary

of the domain and successively "splits" the domain by mapping this exterior Dirichlet data to interior data. At the leaf level, we solve the Poisson problem (3.3). The solve stage algorithm is described in Algorithm 2.

Algorithm 2 Solve stage on a uniformly refined quadtree mesh

```
for τ = N, N − 1, . . . , 0 do ▷ Iterate over quadrants in uniform quadtree
    if τ is a leaf-level patch then
        Solve patch Poisson problem (3.3)
    else ▷ τ has four child patches α, β, γ, ω.
        Use (3.35) to map Dirichlet g_ext on τ to Dirichlet data g_int on the interior
        shared child patch boundaries.
```

3.2.3 Coarse-Fine Interfaces

The merging and splitting processes we have described so far applies only to four sibling patches occupying a coarser level quadrant. For a fully adaptive quadtree, however, not all quadrants are refined to the same level, resulting in coarse-fine interfaces (see Figure 3.2). To accommodate for the coarse-fine interfaces, we coarsen entire patches with coarse-fine interfaces. This approach is similar to the approach used in Babb *et al.* (2018). The mesh is not changed as a result of this process, just the data on a patch, as detailed below.

Prior to merging, we check each of the four children patches to be merged for a patch with data more fine than the other children. We do this by checking the size of the associated grid on that patch. If a coarse-fine interface exists, the data on that patch gets coarsened so as to match its siblings. For the build stage, only $\mathbf{T}^\tau$ and $\mathbf{h}^\tau$ need to be coarsened. We form interpolation matrices $\mathbf{L}_{2,1}$ and $\mathbf{L}_{1,2}$ that map two sides to one or one side to two, respectively. As we employ a cell-centered grid, these

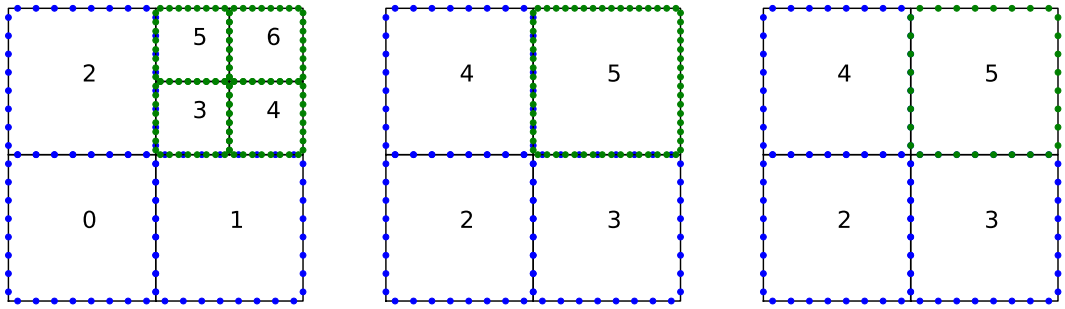


Figure 3.2: Example of a coarse-fine interface merge: (left) patches $6, 7, 8, \& \ 9$ will be merged and result in a coarse-fine interface, (middle) the data on patch 5 is averaged (coarsened), (right) merging $2, 3, 4, \& \ 5$ can continue as detailed in Section 3.2.2.

matrices either average two data points to one, or interpolate one data point to two. Block diagonal versions of these are formed as

$$\mathbf{L}^B_{2,1} = \texttt{BlockDiag}(\{\mathbf{L}_{2,1}, \mathbf{L}_{2,1}, \mathbf{L}_{2,1}, \mathbf{L}_{2,1}\}) \tag{3.36}$$

$$\mathbf{L}^B_{1,2} = \texttt{BlockDiag}(\{\mathbf{L}_{1,2}, \mathbf{L}_{1,2}, \mathbf{L}_{1,2}, \mathbf{L}_{1,2}\}) \tag{3.37}$$

and coarsening $\mathbf{T}$ and $\mathbf{h}$ is done by matrix multiplication

$$\mathbf{T}^{\tau'} = \mathbf{L}^B_{2,1}\mathbf{T}^{\tau}\mathbf{L}^B_{1,2} \tag{3.38}$$

$$\mathbf{h}^{\tau'} = \mathbf{L}^B_{2,1}\mathbf{h}^{\tau} \tag{3.39}$$

where the prime indicates a coarsened version of that data.

A 1-to-4 split with a parent that has data that was coarsened during the 4-to-1 merge (for example, patch 5 from Figure 3.2) needs to be un-coarsened. The requisite data that is un-coarsened during the solve stage is the boundary data, $\mathbf{g}_{ext}$ in Equation

3.35. This is done with the block diagonal interpolation matrices formed above:

$$\mathbf{g}_{ext}^{\tau} = \mathbf{L}_{1,2}^{B}\mathbf{g}_{ext}^{\tau'}. \tag{3.40}$$

3.2.4 Comparison Between 4-to-1 Merging and 2-to-1 Merging

Here, we compare the performance and storage requirements for the 4-to-1 merge outlined in this paper against the 2-to-1 merge presented in Gillman & Martinsson (2014). In Gillman & Martinsson (2014), merging is done between two neighboring patches. To merge a family of four patches, one must do two vertical merges (merge two neighboring patches that lie next to each other in the y-direction) and then one horizontal merge (merge two "tall-skinny" patches that lie next to each other in the x-direction). Thus, computing and storing $\mathbf{T}^{\tau}$ and $\mathbf{S}^{\tau}$ requires three, 2-to-1 merges: two vertical merges and one horizontal merge.

For both approaches, we assume that a patch has M points per side, resulting in M^2 points per patch. We will compare the floating point operations per second (FLOPS), or FLOP count, as well as the memory needed to store the computed matrices. The merge process is seen as an elimination of the points on the interface of neighboring patches. For the 2-to-1 vertical merge, computing $\mathbf{S}$ requires M^3 FLOPS as a linear solve is necessary, and computing $\mathbf{T}$ requires $36M^3$ FLOPS. Storing $\mathbf{S}$ and $\mathbf{T}$ requires $6M^2$ and $36M^2$ numbers, respectively. For the 2-to-1 horizontal merge, computing $\mathbf{S}$ requires $8M^3$ FLOPS and computing $\mathbf{T}$ requires $128M^3$ FLOPS. Storing $\mathbf{S}$ and $\mathbf{T}$ requires $16M^2$ and $64M^2$ numbers, respectively. Thus, for a full 4-to-1 merge via two vertical merges and one horizontal merge, the total FLOP count is $210M^3$,

with storage for $164M^2$ numbers. For the 4-to-1 merge, computing $\mathbf{S}$ requires $64M^3$ FLOPS and computing $\mathbf{T}$ requires $256M^3$ FLOPS. Storing $\mathbf{S}$ and $\mathbf{T}$ requires $32M^2$ and $64M^2$ numbers, respectively. Thus, for a 4-to-1 merge outlined in this paper, the total FLOP count is $320M^3$, with storage for $96M^2$ numbers. Our 4-1 merge requires about 50% more FLOPs than a 2-1 merge on our finite volume mesh. However, the 4-1 requires 70% less storage. We justify the higher FLOP count with the greater ease of implementation of the 4-1 merge over the 2-1, since only one type of merge algorithm is required.

3.3 Implementation Details

In this section, we go over some of the details useful in understanding how the quadtree-adaptive HPS method is implemented. We discuss the patch solver implementation which wraps fast solvers. We provide details related to the mesh library `p4est` which is used as a backend for the quadtree data structure.

3.3.1 Patch Solvers

When solving 3.1 on a single leaf patch, the method used to solve the boundary value problem is called a *patch solver* and we denote this function as `PatchSolver`. The goal of the patch solver is to perform the computations in 3.4. The patch solver takes as input the Dirichlet data on the boundary $\mathbf{g}_{ext}^{\tau}$, the inhomogeneous data $\mathbf{f}^{\tau}$, and some representation of the discretization (i.e., a finite volume grid that corresponds to the patch domain). On output, `PatchSolver` returns the solution of 3.1 on the leaf patch $\mathbf{u}^{\tau}$.

Ideally, the patch solver routine takes advantage of fast and optimized solvers for the boundary value problem 3.1. For this implementation, we wrap the FISHPACK

routines Swarztrauber *et al.* (1999) provided in the FISHPACK90 library Adams *et al.* (2016). FISHPACK solves 3.1 using a cyclic-reduction method, providing a fast and efficient solver for "small" problems. As we use a cell-centered discretization for use with a finite-volume code, we wrap the `hwscrt` method for our `PatchSolver`.

3.3.2 Building Leaf Level Operators

At the leaf level, two operators are required: $\mathbf{S}^\tau$ and $\mathbf{T}^\tau$. The `PatchSolver` takes the role of $\mathbf{S}^\tau$ on the leaf. We explicitly compute the matrix $\mathbf{T}^\tau$ through use of the patch solver. We define a function `BuildD2N` that computes $\mathbf{T}^\tau$ according to 3.10. The Dirichlet-to-Neumann matrix depends solely on the discretization – it is independent of boundary data or inhomogenenous data. Each column of $\mathbf{T}^\tau$ is computed column-by-column by solving 3.1 with columns of the identity matrix $\mathbf{I} \in \mathbb{R}^{4M \times 4M}$ as Dirichlet data $\mathbf{g}^\tau$:

$$\mathrm{col}_j(\mathbf{T}^\tau) = \mathbf{T}^\tau \hat{\mathbf{e}}_j = \frac{2}{h}(\mathbf{I} - \mathbf{G}\mathcal{L}_h^\tau)\hat{\mathbf{e}}_j. \tag{3.41}$$

For a constant coefficient elliptic problem, we can take advantage of symmetry in $\mathbf{T}^\tau$ to reduce the number of calls to the `PatchSolver`. This is due to the limited range of the Dirichlet-to-Neumann operator being a boundary operator that depends solely on the discretization of the elliptic problem. To build $\mathbf{T}^\tau$ using these optimizations, compute the first M columns of $\mathbf{T}^\tau$ and use the following pattern to fill in the remaining

blocks:

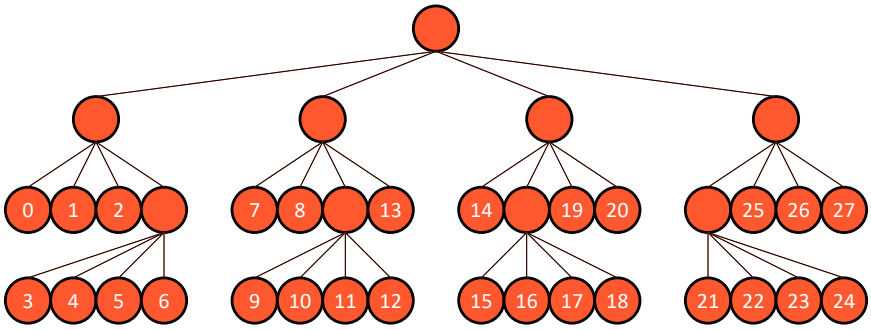
$$\mathbf{T}^{\tau} = \begin{bmatrix} \mathbf{T}_{w,w} & -\mathbf{T}_{w,e} & \mathbf{T}_{w,s} & -\mathbf{T}_{w,n} \\ \mathbf{T}_{w,e} & -\mathbf{T}_{w,w} & \mathbf{T}_{w,n} & \mathbf{T}_{e,n} \\ \mathbf{T}_{w,s} & -\mathbf{T}_{w,n}^{T} & \mathbf{T}_{w,w} & -\mathbf{T}_{w,e} \\ \mathbf{T}_{w,n} & \mathbf{T}_{w,n}^{'} & \mathbf{T}_{w,e} & -\mathbf{T}_{w,w} \end{bmatrix} \tag{3.42}$$

where M is the size of a leaf patch boundary, and $\mathbf{T}_{w,n}^{'}$ is $\mathbf{T}_{w,n}$ with reversed columns: $T_{i,j}^{'} = T_{M-i,j}$.

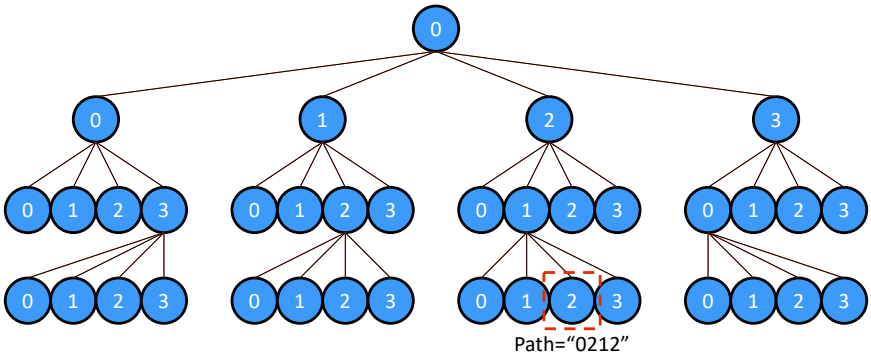
3.3.3 Quadtrees and Adaptive Meshes

As a software library, `p4est` provides a quadtree data structure and functions to construct, store, and iterate over leaf-level quadrants in the quadtree. However, the quadtree-adaptive HPS method requires storage for all nodes in the quadtree; this includes children and parent nodes, not just leaf nodes. This is to store the set of solution operators that act as the matrix factorization of the system matrix. Thus, we require a new data structure that allows for data storage for all nodes in a quadtree. One approach that will later prove to be advantageous for parallel applications is a *path-indexed quadtree* Woodward (1982); Samet (1984). The path of a node is the unique series of directions required to traverse from the root of the tree to the node. The unique path is encoded as a string containing the sequence of nodes visited. Figure 5.2 illustrates the path and this unique encoding for a level 3 node.

A path-indexed quadtree creates storage for all nodes in the tree through the use of a `NodeMap`, equivalent to a C++ `std::map<std::string, Node<T>*>`, where the template parameter `T` corresponds to a user-provided class to be stored at each node. For the quadtree-adaptive HPS method detailed in this paper, this is a class that



(a) Leaf-level indexing of quadtree nodes



(b) Path indexing of quadtree nodes

Figure 3.3: Leaf-indexed vs. path-indexed quadtrees. In (a), only the leaves of a quadtree are indexed and stored. In (b), all nodes of the quadtree are indexed and stored according to their unique path.

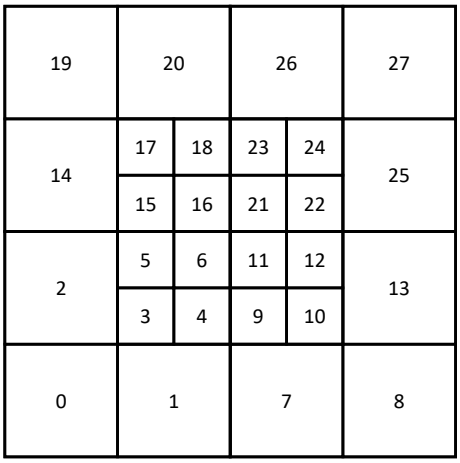


Figure 3.4: A logically square domain is recursively broken into 4 children domains. This mesh represents refinement criteria that refines the center of the domain. The leaf level nodes are indexed according to a leaf-indexed quadtree and follow a space filling curve.

holds patch data (grid information, matrices, and vectors). The path is computed by calling `p4est_quadrant_ancestor_id`. The path-indexed quadtree wraps functions provided by `p4est` to construct and iterate over nodes in the path-indexed quadtree.

Two functions provided by `p4est` allow for a depth-first traversal of a `p4est` quadtree: `p4est_search_all` and `p4est_search_reorder`. `p4est_search_all` performs a top-down search of the quadtree and provides a callback function with access to a `p4est` quadrant data structure. This function is used to initialize the path-indexed quadtree by traversing the quadtree in a depth-first order, computing the path for each node, and allocating memory for a `Node` object in the `NodeMap`. The function `p4est_search_reorder` also does a top-down search, and provides pre- and post-quadrant callback functions for accessing quadrant data.

Wrapping `p4est_search_reorder`, we define the *merge traversal* and the *split traversal* of a quadtree. The merge traversal iterates over a quadtree in a post-order

fashion, visiting children then parents. When visiting a leaf, a leaf callback is called. When visiting parents, a family callback is used that provides access to the parent and the four children nodes. This is to "merge" four children nodes into a parent node, after any leaf data is assigned or computed. The split traversal iterates over a quadtree in a pre-order fashion, with callbacks to families then leaf nodes. This is to provide access to families to "split" one parent node into four children nodes. The leaf callback is done last in the split traversal and is used to compute leaf level data once the entire tree has been traversed. The algorithm for the callback function passed to `p4est_search_reorder` is provided in Algorithm 3.

Algorithm 3 `QuadtreeCallback` Function

Require: Functions `LeafCallback(leaf_node)` & `FamilyCallback(parent_node, children_nodes)`

```
Compute path from p4est_quadrant_ancestor_id
Let node = node_map[path]
if node is a leaf then ▷ Node is a leaf, call leaf callback
    Call LeafCallback(node)
else▷ Node is a branch, get children and call family callback
    for i = 0, 1, 2, 3 do
        children_nodes[i] = map[path + string(i)]
    Call FamilyCallback(node, children_nodes)
```

For the quadtree-adaptive HPS method, the leaf callback for the merge traversal is `BuildD2N`, which solves 3.10 and computes a leaf level Dirichlet-to-Neumann matrix. The family callback wraps the algorithms found in Section 3.2.2, which performs the 4-to-1 merge.

The family callback for the split traversal wraps 3.35, which applies the solution operator to map parent Dirichlet data to children Dirichlet data (the 1-to-4 split). The leaf callback wraps `PatchSolver` to solve 3.1 on a leaf level using fast solution

methods.

3.4 Numerical Results

To test and verify the implementation of the quadtree-adaptive HPS method, we solve two Poisson equations and one Helmholtz equation. For each, we present error, timing, and memory usage results and discuss the performance of our implemented method.

For all of our examples, we discretize the Laplace operator at the patch level using a second-order, 5-point stencil on a finite volume (cell-centered) mesh. The code, along with numerical experiments below, is stored in the GitHub repository EllipticForest Chipman (2023). EllipticForest is written primarily in C++, with wrappers to FORTRAN routines to call FISHPACK and LAPACK (Anderson *et al.* (1999)) for any dense linear algebra operations. The mesh and solution are output into an unstructured VTK mesh file **?** and are visualized with the VisIt software Childs *et al.* (2012). All tests were run on a 2021 MacBook Pro with an M1 Pro CPU and 32 GB of RAM.

3.4.1 Poisson Equation 1

We solve the following boundary value problem:

$$\nabla^2 u(x,y) = -(\sin(x) + \sin(y)) \tag{3.43}$$

on the square domain $\Omega = [-10,10] \times [-10,10]$, subject to Dirichlet boundary conditions $u(x,y) = g(x,y)$ on the boundary which is computed according to the exact solution

$$u_{exact}(x,y) = \sin(x) + \sin(y). \tag{3.44}$$

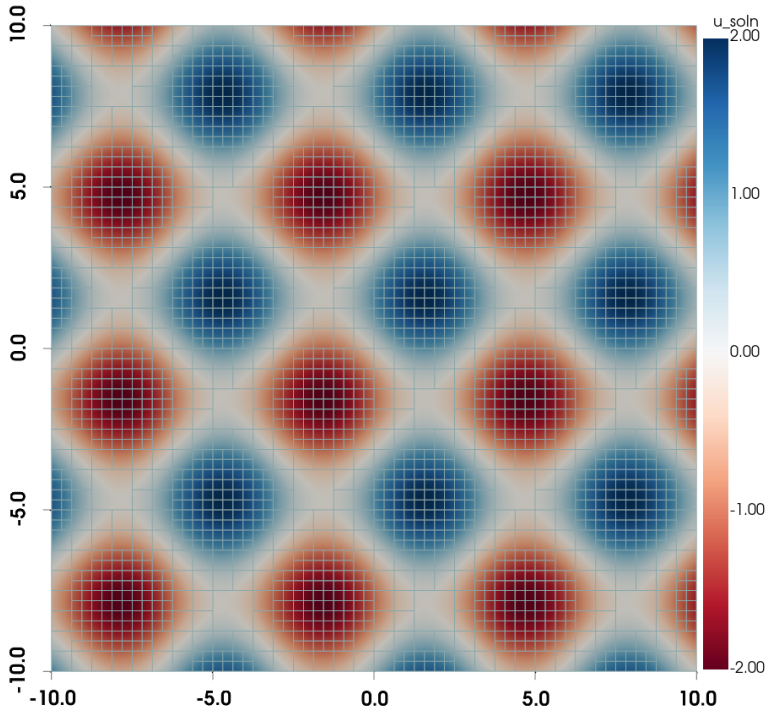


Figure 3.5: The computed solution and mesh for the Poisson problem Section 3.4.1. Patch size for this plot is 16×16 and mesh is refined to level 7. Refinement criteria is based on the magnitude of the right hand side function $f(x, y)$.

For the refinement criteria, we refine according to the right-hand side function $f(x, y)$, which corresponds to the curvature of the solution. We set a refinement threshold of 1.2 and refine a patch when $f(x, y) > 1.2$ for any x, y in a patch. This results in a mesh and solution that can be found in Figure 3.5.

Results and Discussion Tables 3.1 and 3.2 show the error, timing, and memory results for the current implementation on this Poisson equation. Table 3.1 shows results for both a uniformly refined mesh (a mesh without any coarse-fine interfaces or local adaptivity) and results for the adaptive mesh case. For the uniform case,

Table 3.1: Convergence analysis for Poisson's equation. The upper part shows convergence for a uniformly refined mesh, while the lower part shows convergence for an adaptively refined mesh. M is the size of the grid on each leaf patch, $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution for a uniformly refined mesh, DOFs is the total degrees of freedom (i.e., total mesh points), L_∞ error is the infinity norm error, L_∞ order is the infinity norm convergence order, L_1 error is the 1^{st} norm error, and L_1 order is the 1^{st} norm convergence order.

M	$L_{\max}$	R_{eff}	DOFs	L_∞ Error	L_∞ Order	L_1 Error	L_1 Order
16	4	256	65 536	1.11×10^{-3}	2.00	3.59×10^{-4}	2.00
16	5	512	262 144	2.79×10^{-4}	2.00	8.97×10^{-5}	2.00
16	6	1024	1 048 576	6.96×10^{-5}	2.00	2.24×10^{-5}	2.00
16	7	2048	4 194 304	1.74×10^{-5}	2.00	5.61×10^{-6}	2.00
32	4	512	262 144	2.79×10^{-4}	2.00	8.97×10^{-5}	2.00
32	5	1024	1 048 576	6.96×10^{-5}	2.00	2.24×10^{-5}	2.00
32	6	2048	4 194 304	1.74×10^{-5}	2.00	5.61×10^{-6}	2.00
16	4	256	64 000	2.59×10^{-3}	0.78	4.46×10^{-4}	1.69
16	5	512	194 560	6.63×10^{-4}	1.97	1.25×10^{-4}	1.84
16	6	1024	569 344	6.81×10^{-4}	-0.04	1.02×10^{-4}	0.29
16	7	2048	1 984 000	1.71×10^{-4}	1.99	3.76×10^{-5}	1.44
32	4	512	256 000	6.62×10^{-4}	0.75	1.12×10^{-4}	1.68
32	5	1024	784 384	1.67×10^{-4}	1.98	3.11×10^{-5}	1.85
32	6	2048	2 289 664	1.71×10^{-4}	-0.03	2.57×10^{-5}	0.28

we get the expected second order convergence in both the L_∞ and L_1 norms. The adaptive case mostly shows second order convergence, except for a few cases where the refinement between successive levels results in a smaller jump in error than second order provides. Table 3.2 shows timing and memory results for the same case. Here, the difference between the uniform and adaptive case is highlighted. The adaptive case gives a 4.5 times speed up for the build stage, and nearly a 20 times speed up for the solve stage. Memory used to store the quadtree and operators is also significantly reduced with the adaptive case.

Table 3.2: Timing and memory results for Poisson's equation. The upper part shows results for the uniformly refined mesh, while the lower part shows results for the adaptively refined mesh. The results here are for a patch size of 16×16. $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution, DOFs is the total degrees of freedom, T_{build} is the time in seconds for the build stage, T_{upwards} is the time in seconds for the upwards stage, T_{solve} is the time in seconds for the solve stage, and S is the memory storage in megabytes to store the quadtree and all data matrices stored in each node of the quadtree.

$L_{\max}$	R_{eff}	DOFs	T_{build} (sec)	T_{upwards} (sec)	T_{solve} (sec)	S (MB)
4	256	65 536	2.031	0.367	0.211	81.5
5	512	262 144	8.526	1.779	1.074	398.2
6	1024	1 048 576	39.354	9.002	4.259	1881.0
7	2048	4 194 304	172.190	42.035	20.559	8676.2
4	256	64 000	1.502	0.394	0.098	53.0
5	512	194 560	3.719	0.925	0.291	140.1
6	1024	569 344	10.595	2.221	0.478	368.5
7	2048	1 984 000	38.977	8.095	1.839	1443.9

3.4.2 Poisson Equation 2 (Polar-Star Problem)

As another test for Poisson's equation, we solve a "polar star" Poisson problem. The problem is created via a method of manufactured solutions and is engineered to have highly local curvature from the load function. The resulting solution is a collection of polar stars with user specified number of points and radii of curvature. This problem highlights the use of an adaptive mesh to solve the elliptic equation. The exact solution we attempt to reconstruct is the following:

$$u(x,y) = \frac{1}{2}\sum_{i=1}^{N} 1 - \tanh\left(\frac{r(x,y) - r_{0,i}\left(r_{1,i}\cos\left(n\theta(x,y)\right) + 1\right)}{\epsilon}\right) \tag{3.45}$$

Computing the Laplacian analytically yields the right-hand side to Poisson's equation. Thus, the polar star Poisson problem is defined as follows:

$$\nabla^2 u(x,y) = \sum_{i=1}^{N} -\frac{s_{1,i}(x,y) + s_{2,i}(x,y)}{r(x,y)^2} - s_{3,i}(x,y) + s_{4,i}(x,y) \tag{3.46}$$

with

$$\begin{aligned}
s_{1,i}(x,y) &= \frac{p(x,y)^2 \tanh\left(\phi(x,y)\right) \operatorname{sech}^2\left(\phi(x,y)\right)}{\epsilon^2} \\
s_{2,i}(x,y) &= -\frac{n^2 r_{0,i} r_{1,i} \cos(n\theta(x,y)) \operatorname{sech}^2\left(\phi(x,y)\right)}{2\epsilon} \\
s_{3,i}(x,y) &= \frac{\tanh\left(\phi(x,y)\right) \operatorname{sech}^2\left(\phi(x,y)\right)}{\epsilon^2} \\
s_{4,i}(x,y) &= \frac{\operatorname{sech}^2\left(\phi(x,y)\right)}{2r(x,y)\epsilon} \\
p(x,y) &= n r_{0,i} r_{1,i} \sin(n\theta(x,y)) \\
\phi(x,y) &= \frac{r(x,y) - r_{0,i}(r_{1,i}\cos(n\theta(x,y)) + 1)}{\epsilon}
\end{aligned}$$

and where $i = 1, ..., N_{polar}$ and N_{polar} is the number of polar stars. Each polar star has a center (x_0, y_0), inner and outer radii r_0, r_1, and the number of arms per polar star n. The radius and angle have the standard polar transforms:

$$r(x,y) = \sqrt{(x-x_0)^2 + (y-y_0)^2} \tag{3.47}$$

$$\theta(x,y) = \tan^{-1}\left(\frac{y-y_0}{x-x_0}\right) \tag{3.48}$$

Table 3.3 has the parameters used in this case study. Figure 3.6 shows the resulting mesh and solution. Table 3.4 shows the error, timing, and memory results for the polar star Poisson problem.

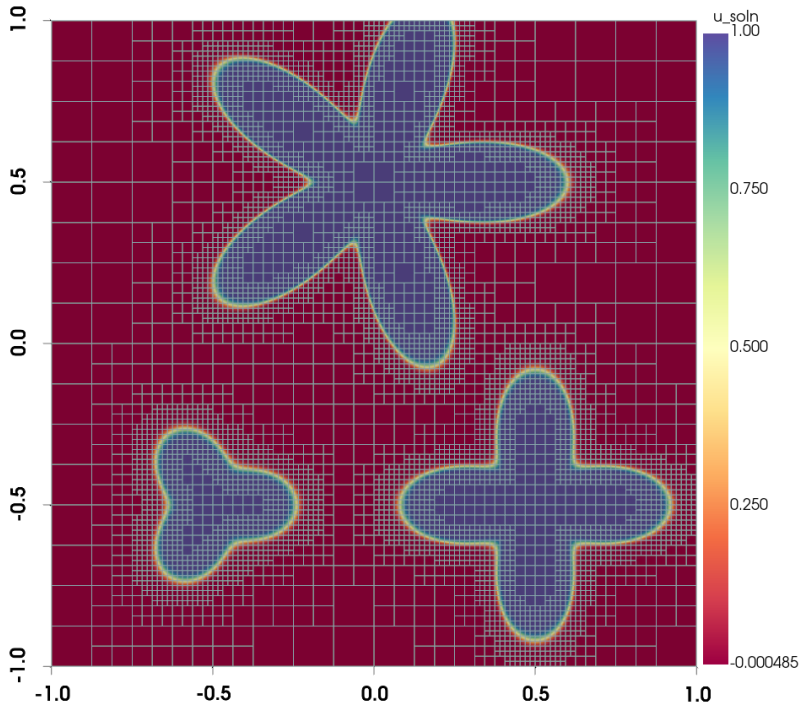


Figure 3.6: The computed solution and mesh for the Polar Star Poisson problem. Each patch has a 16×16 cell-centered grid. The mesh is refined according to the right-hand side and is refined with 8 levels of refinement.

Table 3.3: Polar Star Poisson Problem Parameters

i	x_0	y_0	r_0	r_1	n
1	-0.5	-0.5	0.2	0.3	3
2	0.5	-0.5	0.3	0.4	4
3	0	0.5	0.4	0.5	5

Results and Discussion The polar star Poisson problem is able to highlight the benefits of an adaptive mesh. As shown in Figure 3.6, there is room for significant speed up on the areas outside each polar star. The localized curvature is sufficiently captured by this adaptive scheme. Table 3.4 show the error analysis for this problem. Because of the large curvature at the boundaries of a polar star and our use of a second-order accurate solver, the L_∞ error is larger than the first Poisson equation from Section 3.4.1, but we can still achieve 6 digits of accuracy in the L_1 norm. The timing results shown in Table 3.5 show significant speed up between the uniform and adaptive implementations. The build time has a 5.5 times speed up, the upwards stage has a 5.8 times speed up, and the solve stage has a 17 times speed up. Plus, the memory for the adaptive case is about $1/8th$ of the memory required for the uniform case.

3.4.3 Helmholtz's Equation

We now seek to solve the boundary value problem

$$\nabla^2 u(x, y) + \lambda u(x, y) = f(x, y), \tag{3.49}$$

on the square domain $\Omega = [-0.5, 0.5] \times [-0.5, 0.5]$ and subject to Dirichlet boundary conditions which are supplied via the exact solution discussed below.

Table 3.4: Convergence analysis for the Polar Star Poisson problem. The upper part shows convergence for a uniformly refined mesh, while the lower part shows convergence for an adaptively refined mesh. M is the size of the grid on each leaf patch, $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution for a uniformly refined mesh, DOFs is the total degrees of freedom (i.e., total mesh points), L_∞ error is the infinity norm error, L_∞ order is the infinity norm convergence order, L_1 error is the 1^{st} norm error, and L_1 order is the 1^{st} norm convergence order.

M	$L_{\max}$	R_{eff}	DOFs	L_∞ Error	L_∞ Order	L_1 Error	L_1 Order
16	4	256	65 536	1.56	3.04	1.65×10^{-1}	4.25
16	5	512	262 144	2.80×10^{-2}	5.80	9.85×10^{-4}	7.39
16	6	1024	1 048 576	5.25×10^{-3}	2.41	7.40×10^{-5}	3.73
16	7	2048	4 194 304	1.28×10^{-3}	2.03	1.84×10^{-5}	2.01
32	3	256	65 536	1.56	3.04	1.65×10^{-1}	4.25
32	4	512	262 144	2.80×10^{-2}	5.80	9.85×10^{-4}	7.39
32	5	1024	1 048 576	5.25×10^{-3}	2.41	7.40×10^{-5}	3.73
32	6	2048	4 194 304	1.28×10^{-3}	2.03	1.84×10^{-5}	2.01
16	4	256	50 944	1.56	3.04	1.65×10^{-1}	4.25
16	5	512	171 520	7.12×10^{-2}	4.45	1.03×10^{-3}	7.32
16	6	1024	476 416	9.15×10^{-2}	-0.36	1.79×10^{-4}	2.53
16	7	2048	1 587 712	2.82×10^{-2}	1.70	6.33×10^{-5}	1.50
32	3	256	65 536	1.56	3.04	1.65×10^{-1}	4.25
32	4	512	203 776	2.80×10^{-2}	5.80	9.84×10^{-4}	7.39
32	5	1024	701 440	2.34×10^{-2}	0.26	8.15×10^{-5}	3.59
32	6	2048	1 921 024	2.82×10^{-2}	-0.27	3.10×10^{-5}	1.39

Table 3.5: Timing and memory results for the Polar Star Poisson problem. The upper part shows results for the uniformly refined mesh, while the lower part shows results for the adaptively refined mesh. The results here are for a patch size of 16×16. $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution, DOFs is the total degrees of freedom, T_{build} is the time in seconds for the build stage, T_{upwards} is the time in seconds for the upwards stage, T_{solve} is the time in seconds for the solve stage, and S is the memory storage in megabytes to store the quadtree and all data matrices stored in each node of the quadtree.

$L_{\max}$	R_{eff}	DOFs	T_{build} (sec)	T_{upwards} (sec)	T_{solve} (sec)	S (MB)
4	256	65 536	1.61	0.42	0.21	81.55
5	512	262 144	7.77	1.97	0.97	398.23
6	1024	1 048 576	35.67	9.24	4.10	1881.01
7	2048	4 194 304	165.94	42.46	19.35	8676.20
4	256	50 944	0.93	0.24	0.08	33.52
5	512	171 520	3.33	0.79	0.14	115.86
6	1024	476 416	9.07	1.94	0.30	295.11
7	2048	1 587 712	30.78	7.23	1.12	1076.81

To provide an analytical solution to compare against, we use a problem provided in Cheng *et al.* (2006) where they solve Helmholtz's equation on an adaptive mesh. The manufactured solution is

$$u(\mathbf{x}) = \sum_{i=1}^{3} e^{-\alpha|\mathbf{x}-\mathbf{x}_i|^2} \tag{3.50}$$

with $\lambda = 0.01$, $\alpha = 50$, $\mathbf{x}_1 = (0.1, 0.1)$, $\mathbf{x}_2 = (0, 0)$, and $\mathbf{x}_3 = (-0.15, 0.1)$. The right-hand side is computed analytical via the Mathematica software **?** and is used as the refinement criteria. We set the threshold for refinement to 60. The solution and right-hand side function for 16×16 patches are plotted in Figure 3.7 and Figure 3.8.

Results and Discussion As demonstrated with this example, the implemented HPS solves Helmholtz equations as well with the expected second order accuracy. See

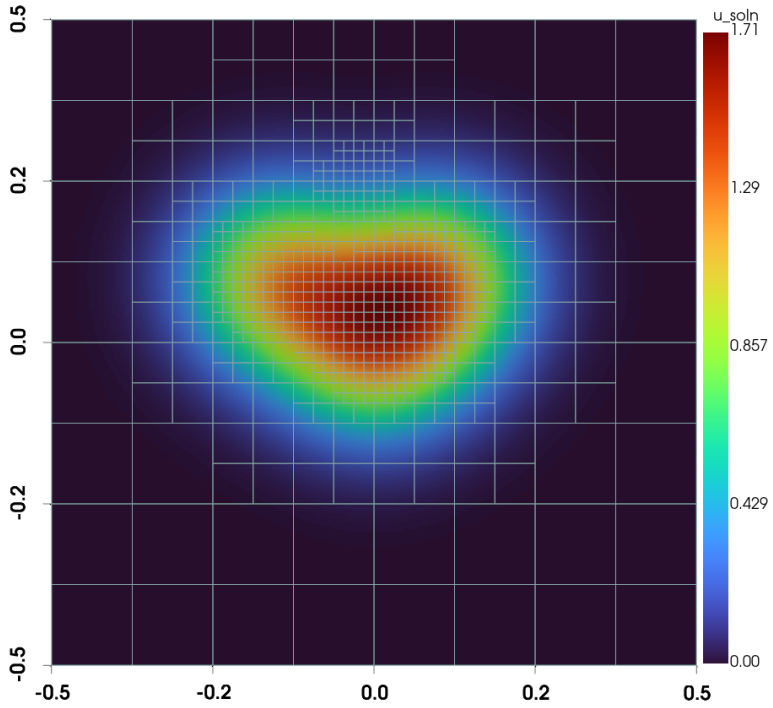


Figure 3.7: Mesh and plot of the solution to the Helmholtz problem in Section 3.4.3.

Table 3.6 for error and convergence analysis. The adaptive mesh is able to successfully capture the curvature and reduce the work required to solve this equation. Table 3.7 show timing and memory usage results, which show significant speedup due to good mesh adaptation. We achieve a 17 times speedup for the build and upwards stage and a 57 times speedup for the solve stage. Memory is also impressive for this particular problem, with solving the same problem with similar error with 4.6% of the required memory.

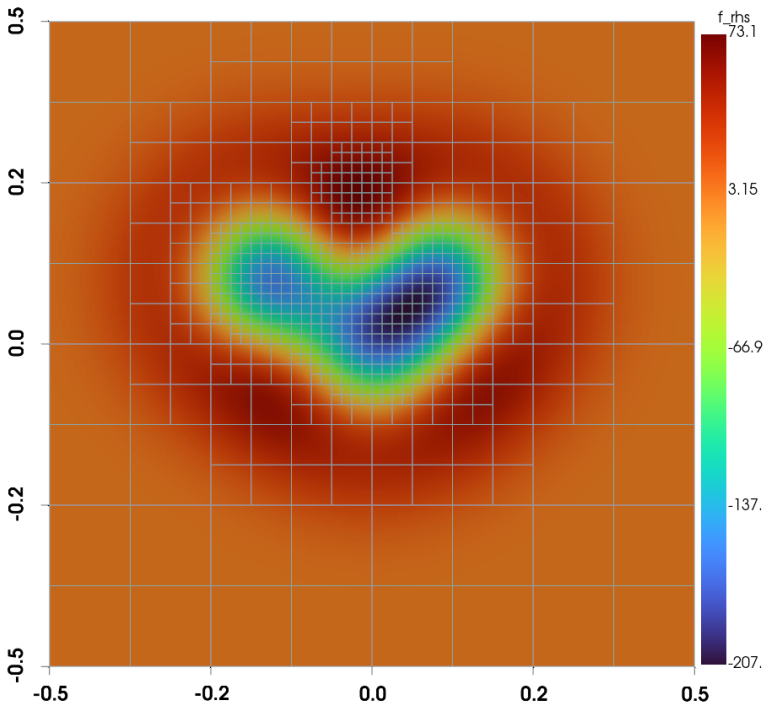


Figure 3.8: Mesh and right-hand side of the solution to the Helmholtz problem in Section 3.4.3.

Table 3.6: Convergence analysis for the Helmholtz problem. The upper part shows convergence for a uniformly refined mesh, while the lower part shows convergence for an adaptively refined mesh. M is the size of the grid on each leaf patch, $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution for a uniformly refined mesh, DOFs is the total degrees of freedom (i.e., total mesh points), L_∞ error is the infinity norm error, L_∞ order is the infinity norm convergence order, L_1 error is the 1^{st} norm error, and L_1 order is the 1^{st} norm convergence order.

M	$L_{\max}$	R_{eff}	DOFs	L_∞ Error	L_∞ Order	L_1 Error	L_1 Order
16	3	128	16 384	8.12×10^{-4}	2.00	8.79×10^{-5}	2.00
16	4	256	65 536	2.03×10^{-4}	2.00	2.20×10^{-5}	2.00
16	5	512	262 144	5.07×10^{-5}	2.00	5.49×10^{-6}	2.00
16	6	1024	1 048 576	1.27×10^{-5}	2.00	1.37×10^{-6}	2.00
16	7	2048	4 194 304	3.17×10^{-6}	2.00	3.43×10^{-7}	2.00
32	3	256	65 536	2.03×10^{-4}	2.00	2.20×10^{-5}	2.00
32	4	512	262 144	5.07×10^{-5}	2.00	5.49×10^{-6}	2.00
32	5	1024	1 048 576	1.27×10^{-5}	2.00	1.37×10^{-6}	2.00
32	6	2048	4 194 304	3.17×10^{-6}	2.00	3.43×10^{-7}	2.00
16	3	128	8704	1.36×10^{-3}	1.26	1.62×10^{-4}	1.12
16	4	256	22 528	1.65×10^{-3}	-0.28	8.46×10^{-5}	0.94
16	5	512	54 784	2.39×10^{-3}	-0.53	3.39×10^{-5}	1.32
16	6	1024	163 072	5.95×10^{-4}	2.00	1.31×10^{-5}	1.37
16	7	2048	485 632	7.32×10^{-4}	-0.30	1.91×10^{-5}	-0.54
32	3	256	34 816	3.39×10^{-4}	1.26	4.07×10^{-5}	1.11
32	4	512	90 112	4.02×10^{-4}	-0.25	2.13×10^{-5}	0.93
32	5	1024	222 208	5.86×10^{-4}	-0.54	8.31×10^{-6}	1.36
32	6	2048	664 576	1.51×10^{-4}	1.96	3.72×10^{-6}	1.16

Table 3.7: Timing and memory results for the Helmholtz problem. The upper part shows results for the uniformly refined mesh, while the lower part shows results for the adaptively refined mesh. The results here are for a patch size of 16×16. $L_{\max}$ is the maximum level of refinement, R_{eff} is the effective resolution, DOFs is the total degrees of freedom, T_{build} is the time in seconds for the build stage, T_{upwards} is the time in seconds for the upwards stage, T_{solve} is the time in seconds for the solve stage, and S is the memory storage in megabytes to store the quadtree and all data matrices stored in each node of the quadtree.

$L_{\max}$	R_{eff}	DOFs	T_{build} (sec)	T_{upwards} (sec)	T_{solve} (sec)	S (MB)
3	128	16 384	0.30	0.08	0.05	15.88
4	256	65 536	1.49	0.40	0.20	81.55
5	512	262 144	7.22	1.88	0.91	398.23
6	1024	1 048 576	33.90	8.80	3.90	1881.01
7	2048	4 194 304	159.02	42.57	18.87	8676.20
3	128	8704	0.13	0.03	0.01	4.63
4	256	22 528	0.36	0.08	0.02	13.38
5	512	54 784	0.87	0.20	0.04	28.44
6	1024	163 072	2.89	0.72	0.13	116.81
7	2048	485 632	9.31	2.39	0.33	395.93

3.5 Conclusion

We have demonstrated a new implementation of the Hierarchical Poincaré-Steklov method on a quadtree-adaptive mesh. We outlined the key differences between a binary tree and quadtree structure, as well as the linear algebra associated with a quadtree implementation. Our novel full quadtree indexing allows for efficient storage of the data matrices needed in the HPS method.

Our numerical experiments show that the quadtree-adaptive HPS method is fast and efficient in terms of time, error, and data storage. We solved three elliptic partial differential equations to demonstrate the correctness and efficiency of our method. This kind of implementation is well-suited for an adaptive mesh framework such as `p4est`. Indeed, we have shown that the quadtree-adaptive HPS method can solve the elliptic problems to similar error with up to 17 times speedup in the build stage and up to 57 times speedup in the solve stage. The speedup depends on how well the mesh is able to adapt to the curvature of the solution. Memory use is of particular interest when solving elliptic PDEs with a direct method, and we have demonstrated that this is a highly efficient implementation.

A major advantage of this direct solver for elliptic problems is the ability to pre-compute the set of solution operators needed to solve the problem. Once the set of solution operators is known, a solve step can be done in linear time. This is especially powerful for time-stepping problems where one can pre-compute the solution operators, then apply them at subsequent time steps. Further development of this implementation is in progress to couple this elliptic solver to a hyperbolic finite volume solver through an operator splitting approach. In addition, a fully parallel implementation is being developed for use on large supercomputers to solve coupled

hyperbolic/elliptic partial differential equations.

CHAPTER 4:

AN ADAPTIVE REBUILD IMPROVEMENT TO THE QUADTREE-ADAPTIVE HPS METHOD

4.1 Introduction

4.2 Mathematical Theory for the Adaptive Rebuild

4.3 The Adaptive-Rebuild Algorithm

4.4 Numerical Experiments

4.5 Conclusion

CHAPTER 5:
A PARALLEL IMPLEMENTATION OF THE QUADTREE-ADAPTIVE HPS METHOD

5.1 Introduction

Parallel linear solvers aim to solve large systems of equations on multi-core and/or multi-CPU computer architectures. From desktop workstations with a single CPU with many cores, to supercomputers with hundreds of racks of nodes yielding thousands of CPUs, modern high performance computing systems provide significant compute power for numerical methods. The algorithms employed across these machines take advantage of massive compute power by distributing the workload across all compute resources. Developing parallel algorithms for solving partial differential equations on supercomputers has an active area of research, even prior to the deployment of modern clusters Ortega & Voigt (1985).

Domain decomposition methods are popular approaches to solving elliptic partial differential equations in parallel Smith (1997). By partitioning the domain, each compute unit can independently work towards the solution of the global problem by solving the problem on a subset of the entire domain. The global solution can be constructed by equating the solution and/or fluxes across partition boundaries. Communication between compute units is required during this reconstruction. Optimizing

the communication between compute units, often through asynchronous techniques Glusa *et al.* (2020) or by reducing the amount of communication, results in more efficient implementations. The memory bandwidth between compute units is often one to several orders of magnitude smaller compared to the bandwidth on the compute unit.

With a partitioned domain, the goal is to solve the elliptic PDE using numerical methods such as finite difference, finite volume, finite element, and more. These discretizations lead to a linear system that must be solved. Ideally, the solvers take advantage of the sparsity and structure of the associated linear system. Parallel linear solvers using iterative or direct methods have been successfully developed and implemented. Parallel iterative methods include GMRES Saad & Schultz (1986), the conjugate gradient method Hestenes *et al.* (1952), the algebraic multigrid method Yang *et al.* (2002), and the geometric multigrid method Sundar *et al.* (2012) (with a good overview provided in Chow *et al.* (2006)). Parallel direct methods include matrix factorization methods like LU-factorization, Cholesky factorization and the spectral value decomposition Donfack *et al.* (2015); Demmel *et al.* (1999); Gupta *et al.* (1997).

Several, large-scale and open-source codes currently exist to solve linear systems formed from elliptic PDEs. The hypre library Falgout & Yang (2002) features scalable preconditioners for parallel multigrid methods. The SuperLU library Li (2005) is a general purpose library for solving sparse linear systems using direct methods. Additional codes like PETSc Balay *et al.* (2023), FLASH-X Dubey *et al.* (2022), and AMReX Zhang *et al.* (2019), contain iterative and direct solvers that also work with adaptive mesh refinement. The ForestClaw code Calhoun & Burstedde (2017) coupled with the ThunderEgg repository Aiton *et al.* (2022) implements hyperbolic and elliptic

solvers for finite volume meshes on adaptively refined quadtrees and octrees provided from `p4est`. `p4est` Burstedde *et al.* (2011); Burstedde (2020) is a highly scalable AMR code that provides quadtree and octree data structures for users to build on top of. The EllipticForest library Chipman (2024) contains an implementation of the quadtree-adaptive HPS method, including the parallel algorithms outlined in this paper. Another code of interest is the ButterflyPACK library Liu (2018) that solves large-scale dense systems with off-diagonal, low-rank structure like the matrices formed in the HPS method.

The Hierarchical Poincaré-Steklov (HPS) method Martinsson (2015); Gillman & Martinsson (2014) is a matrix-free, direct method for solving elliptic PDEs. The goal is to build up a factorization by successively merging subdomains via a class of Poincaré-Steklov operators Quarteroni & Valli (1991) called Dirichlet-to-Neumann operators. It is a domain decomposition method that was originally inspired by the partitioning scheme of nested dissection George (1973); Lipton *et al.* (1979). The Dirichlet-to-Neumann operators have off-diagonal, low-rank structure that can be exploited to achieve linear $\mathcal{O}(N)$ complexity in the factorization stage Gillman & Martinsson (2014). While relatively new, the HPS method has been applied to solve 3D elliptic PDEs Hao & Martinsson (2016), has been coupled with the spectral element method Fortunato *et al.* (2020), and implemented on adaptive meshes Babb *et al.* (2018); Geldermans & Gillman (2019); Chipman *et al.* (2024). Recently, the HPS method has also been implemented in parallel, targeting shared-memory machines Beams *et al.* (2020), distributed-memory machines Yesypenko & Martinsson (2022b), and GPU devices Yesypenko & Martinsson (2022a).

The quadtree-adaptive HPS method presented in Chipman *et al.* (2024) is an

implementation of the HPS method on adaptively refined quadtree meshes provided from `p4est` Burstedde *et al.* (2011); Burstedde (2020). `p4est` provides efficiently partitioned quadtree data structures for the quadtree-adaptive HPS method. The novelty of this paper is the porting of the quadtree-adaptive HPS method in parallel, building on top of the parallel infrastructure provided from `p4est`. This paper will present how the leaf-indexed quadtree data structure provided from `p4est` is wrapped, the communication patterns associated with building the matrix-free factorization, and scaling analysis and discussion.

As an overview for this paper, Section 5.2 provides an overview of the quadtree-adaptive HPS method, including the problem description and algorithms. Section 5.3 describes the parallel implementation and details the data structures and algorithms necessary for distributed-memory parallelism. Section 5.4 contains the results and a discussion of the implementation presented herein.

5.2 Overview of the Quadtree-Adaptive HPS Method

In this section, we provide an overview of the quadtree-adaptive HPS method that is outlined in more detail in Chipman *et al.* (2024). This is to provide context as well as demonstrate where we can implement distributed memory parallelism.

5.2.1 Problem Statement and Domain Representation

The problem we wish to solve is the following variable coefficient elliptic PDE subject to Dirichlet or Neumann boundary conditions:

$$\nabla \cdot (\beta(\mathbf{x})\nabla u(\mathbf{x})) + \lambda(\mathbf{x})u(\mathbf{x}) = f(\mathbf{x}), \mathbf{x} \in \Omega \subset \mathcal{R}^2 \tag{5.1}$$

$$u(\mathbf{x}) = g(\mathbf{x}), \mathbf{x} \in \Gamma_D \subset \Omega \tag{5.2}$$

$$\frac{\partial u}{\partial n}\Big|_{\mathbf{x}} = v(\mathbf{x}), \mathbf{x} \in \Gamma_N \subset \Omega. \tag{5.3}$$

The domain Ω is partitioned into a composite collection of subdomains Ω_i such that $\Omega = \cup_{i=1}^{N} \Omega_i$. This is displayed in Figure 5.1. These subdomains are organized into a *leaf-indexed quadtree*

$$\mathcal{Q}_L = \{\Omega_i | i = 1, \ldots, N_L\}, \tag{5.4}$$

where N_L is the number of leaf nodes. A representation of the mesh in Figure 5.1 as a leaf-indexed quadtree can be found in Figure 5.2a. Building up $\mathcal{Q}_L$ is done by refining a logically square domain recursively into children patches according to some initial refinement criteria that is problem dependent or user-supplied.

As detailed in Chipman *et al.* (2024), we also need to create a *path-indexed quadtree* that has data storage for all nodes in the quadtree. We define the path-indexed quadtree formally as

$$\mathcal{Q}_P = \{\Omega^\tau | \tau = 1, \ldots, N_P\}, \tag{5.5}$$

where τ is a key that represents the path of the node and N_P is the number of paths. A path-indexed quadtree representing the mesh in Figure 5.1 is found in Figure 5.2b. The path-indexed quadtree $\mathcal{Q}_P$ is built by iterating over $\mathcal{Q}_L$ and creating storage for

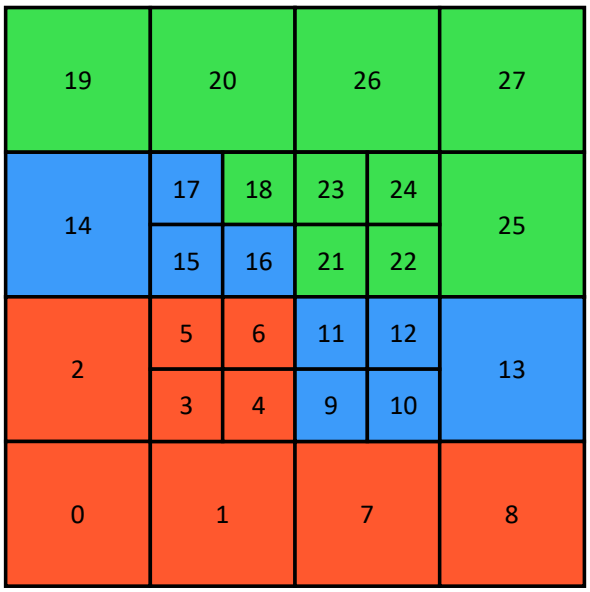


Figure 5.1: An example of an adaptive, quadtree mesh that is refined around the center of the domain. The colors denote different ranks and the numbers indicate the leaf-index of the quadrant.

leaf nodes and their associated ancestors.

5.2.2 Building the Set of Solution Operators

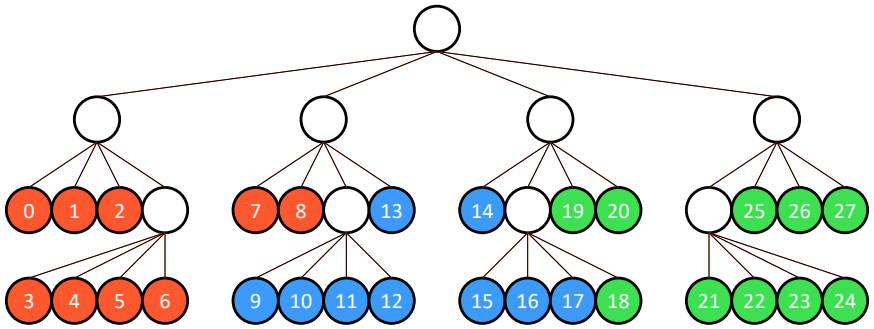
Given $\mathcal{Q}_P$, the quadtree-adaptive HPS method builds up a set of solution operators

$$\mathcal{S} = \{\mathbf{S}^\tau, \mathbf{w}^\tau \ \forall \ \tau \in \mathcal{Q}_P\} \tag{5.6}$$

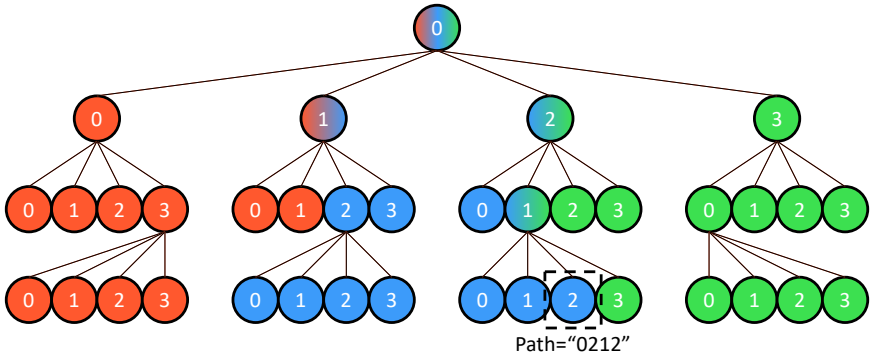
such that

$$\mathbf{u}^\tau = \mathbf{S}^\tau \mathbf{g}^\tau + \mathbf{w}^\tau, \ \forall \ \tau \in \mathcal{Q}_P. \tag{5.7}$$

The operators $\mathbf{S}^\tau$ and $\mathbf{w}^\tau$ are known as the solution matrix and inhomogeneous-update vector, respectively. The vector $\mathbf{g}^\tau$ represents the provided or computed Dirichlet data



(a) Leaf-level indexing of quadtree nodes



(b) Path indexing of quadtree nodes

Figure 5.2: Leaf-indexed vs. path-indexed quadtrees. Both trees represent the mesh found in Figure 5.1. The colors denote which rank owns each node. In (a), only the leaves of a quadtree are indexed and stored. In (b), all nodes of the quadtree are indexed and stored according to their unique path. Note that the nodes in (b) that are colored with a gradient (i.e., "0", "01", "02", "021") are owned by multiple ranks.

for a given patch τ. 5.7 is used to map the known solution data on the exterior of a patch (i.e., $\mathbf{g}^\tau$) to the interior of the patch, which is denoted by $\mathbf{u}^\tau$.

Building up the set of solution operators $\mathcal{S}$ is the goal of the quadtree-adaptive HPS method. This is done by successively merging four children-level operators to form parent-level operators. The operators to be computed and merged are the Dirichlet-to-Neumann matrix $\mathbf{T}^\tau$ and the inhomogeneous-flux vector $\mathbf{h}^\tau$. Starting at the leaf-level, $\mathbf{T}^\tau$ and $\mathbf{h}^\tau$ are built using a *patch solver* function that solves 5.1 on a single patch Ω_i.

Once computed for the four children patches, the parent-level data can be computed. We denote children patches as α, β, γ, and ω. The merged Dirichlet-to-Neumann matrix is decomposed into

$$\mathbf{T}^\tau = \mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C} \tag{5.8}$$

where $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{D}$ are built from blocks of the children-level Dirichlet-to-Neumann matrices:

$$\mathbf{A} = \begin{bmatrix} \mathbf{T}^\alpha_{\tau\tau} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{T}^\beta_{\tau\tau} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{T}^\gamma_{\tau\tau} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{T}^\omega_{\tau\tau} \end{bmatrix} \tag{5.9}$$

$$\mathbf{B} = \begin{bmatrix} \mathbf{T}^{\alpha}_{\gamma\tau} & \mathbf{0} & \mathbf{T}^{\alpha}_{\beta\tau} & \mathbf{0} \\ \mathbf{0} & \mathbf{T}^{\beta}_{\omega\tau} & \mathbf{T}^{\beta}_{\alpha\tau} & \mathbf{0} \\ \mathbf{T}^{\gamma}_{\alpha\tau} & \mathbf{0} & \mathbf{0} & \mathbf{T}^{\gamma}_{\omega\tau} \\ \mathbf{0} & \mathbf{T}^{\omega}_{\beta\tau} & \mathbf{0} & \mathbf{T}^{\omega}_{\gamma\tau} \end{bmatrix} \tag{5.10}$$

$$\mathbf{C} = \begin{bmatrix} \mathbf{T}^{\alpha}_{\tau\gamma} & \mathbf{0} & \mathbf{T}^{\gamma}_{\tau\alpha} & \mathbf{0} \\ \mathbf{0} & \mathbf{T}^{\beta}_{\tau\omega} & \mathbf{0} & \mathbf{T}^{\omega}_{\tau\beta} \\ \mathbf{T}^{\alpha}_{\tau\beta} & \mathbf{T}^{\beta}_{\tau\alpha} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{T}^{\gamma}_{\tau\omega} & \mathbf{T}^{\omega}_{\tau\gamma}, \end{bmatrix} \tag{5.11}$$

$$\mathbf{D} = \begin{bmatrix} \mathbf{T}^{\alpha}_{\gamma\gamma} + \mathbf{T}^{\gamma}_{\alpha\alpha} & \mathbf{0} & \mathbf{T}^{\alpha}_{\beta\gamma} & \mathbf{T}^{\gamma}_{\omega\alpha} \\ \mathbf{0} & \mathbf{T}^{\beta}_{\omega\omega} + \mathbf{T}^{\omega}_{\beta\beta} & \mathbf{T}^{\beta}_{\alpha\omega} & \mathbf{T}^{\omega}_{\gamma\beta} \\ \mathbf{T}^{\alpha}_{\gamma\beta} & \mathbf{T}^{\beta}_{\omega\alpha} & \mathbf{T}^{\alpha}_{\beta\beta} + \mathbf{T}^{\beta}_{\alpha\alpha} & \mathbf{0} \\ \mathbf{T}^{\gamma}_{\alpha\omega} & \mathbf{T}^{\omega}_{\beta\gamma} & \mathbf{0} & \mathbf{T}^{\gamma}_{\omega\omega} + \mathbf{T}^{\omega}_{\gamma\gamma} \end{bmatrix}. \tag{5.12}$$

The subscripts of each Dirichlet-to-Neumann matrix $\mathbf{T}^{k}_{ij}$ indicate a mapping from Dirichlet data from the edge i to Neumann data on edge j of patch k.

The merged inhomogeneous-flux vector $\mathbf{h}^{\tau}$ is computed as

$$\mathbf{h}^{\tau} = -\mathbf{h}_{\text{ext}} + \mathbf{B}\mathbf{D}^{-1}\Delta\mathbf{h} \tag{5.13}$$

where $\mathbf{h}_{\text{ext}}$ corresponds to the exterior of Ω_τ and

$$\Delta\mathbf{h} = \begin{bmatrix} \mathbf{h}_\gamma^\alpha + \mathbf{h}_\alpha^\gamma \\ \mathbf{h}_\omega^\beta + \mathbf{h}_\beta^\omega \\ \mathbf{h}_\beta^\alpha + \mathbf{h}_\alpha^\beta \\ \mathbf{h}_\omega^\gamma + \mathbf{h}_\gamma^\omega \end{bmatrix}. \tag{5.14}$$

The merged, parent-level solution matrix and inhomogeneous-update vector are computed from children-level patches as

$$\mathbf{S}^\tau = -\mathbf{D}^{-1}\mathbf{C} \tag{5.15}$$

$$\mathbf{w}^\tau = -\mathbf{D}^{-1}\Delta\mathbf{h}. \tag{5.16}$$

5.2.3 Stages of the Quadtree-Adaptive HPS Method

Algorithmically, the quadtree-adaptive HPS method is implemented with three stages: a build stage, an upwards stage, and a solve stage. The build stage computes $\mathbf{T}^\tau$ and $\mathbf{S}^\tau \ \forall \ \tau \in \mathcal{Q}_P$. The upwards stage computes $\mathbf{h}^\tau$ and $\mathbf{w}^\tau \ \forall \ \tau \in \mathcal{Q}_P$. Building up the set of solution operators $\mathcal{S}$ is split into the build and the upwards stage to reduce redundant calculations when solving 5.1 with a homogeneous RHS, or $f(\mathbf{x}) = 0$. The solve stage applies the solution operators $\mathbf{S}^\tau$ and $\mathbf{w}^\tau$ from the set $\mathcal{S}$ to supplied Dirichlet boundary data $\mathbf{g}^\tau$ on Ω_D according to 5.7.

In each of the stages, there are two traversals of the quadtree $\mathcal{Q}_P$: a leaf traversal and a family traversal. A leaf traversal visits each of the leaf nodes in the leaf-indexed or path-indexed quadtree (the leaf nodes for both trees are identical). A family traversal visits each family of the quadtree, where a family consists of four children

nodes and a parent node. Family traversals can be done in a pre-order fashion (start at the root and visit families before moving down the tree) or in a post-order fashion (start at the leaf nodes and visit families before moving up the tree). Details on how to traverse a leaf- or path-indexed quadtree will be provided in Section 5.3.2.

The Build Stage

The leaf traversal of the build stage computes $\mathbf{T}^\tau$ for all leaf-level patches. As outlined in Chipman *et al.* (2024), this is done by calling a patch solver that employs fast methods to solve 5.1. 5.1 is solved with unit potentials placed on each of the cell centers on the boundary of the leaf-level patch to compute the columns of $\mathbf{T}^\tau$.

The family traversal of the build stage is outlined in Algorithm 4 and is done in a post-order fashion. The goal of the 4-to-1 merge is to compute the parent-level Dirichlet-to-Neumann matrix and the solution operator from the four children patches. This is done with 5.8 and 5.15.

Algorithm 4 `Merge4To1` Function (Build Stage Family Callback)

Require: $\mathbf{T}^\alpha$, $\mathbf{T}^\beta$, $\mathbf{T}^\gamma$, $\mathbf{T}^\omega$
  for $i = \alpha, \beta, \gamma, \omega$ **do**
    `tag` = `TagForCoarsening`(i)
    if `tag` **then**
      Coarsen: $\mathbf{T}^i = \mathbf{L}_{2,1}\mathbf{T}^i\mathbf{L}_{1,2}$
  Compute matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ according to 5.9 - 5.12.
  Compute $\mathbf{S}^\tau = \mathbf{D}^{-1}\mathbf{C}$
  Compute $\mathbf{T}^\tau = \mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C}$

The Upwards Stage

The leaf traversal of the upwards stage computes $\mathbf{h}^\tau$ for all leaf-level patches. The is done by calling the same routine used to build $\mathbf{T}^\tau$ on leaf-level patches in the build

stage, while supplying zero Dirichlet boundary conditions: $\mathbf{g}^\tau = \mathbf{0}$.

The family traversal is outlined in Algorithm 5 and is also done in a post-order fashion. The upwards 4-to-1 merge computes the merged inhomogeneous-flux vector $\mathbf{h}^\tau$ via 5.13 and the inhomogeneous-update vector $\mathbf{w}^\tau$ via 5.16.

Algorithm 5 `Upwards4To1` Function (Upwards Stage Family Callback)

Require: $\mathbf{T}^\alpha$, $\mathbf{T}^\beta$, $\mathbf{T}^\gamma$, $\mathbf{T}^\omega$, $\mathbf{h}^\alpha$, $\mathbf{h}^\beta$, $\mathbf{h}^\gamma$, $\mathbf{h}^\omega$
  for $i = \alpha, \beta, \gamma, \omega$ **do**
    `tag` = `TagForCoarsening`(i)
    if `tag` **then**
      Coarsen: $\mathbf{h}^i = \mathbf{L}_{2,1}\mathbf{h}^i$
  Compute matrices $\mathbf{B}, \mathbf{D}$ according to 5.10 and 5.12
  Compute $\Delta\mathbf{h}$ according to 5.14
  Compute $\mathbf{h}^\tau = \mathbf{h}^\tau + \mathbf{B}\mathbf{w}^\tau$
  Compute $\mathbf{w}^\tau = -\mathbf{D}^{-1}\Delta\mathbf{h}$.

The Solve Stage

The goal of the solve stage is to apply the now computed solution matrix $\mathbf{S}^\tau$ and inhomogeneous-update vector $\mathbf{w}^\tau$ to provided Dirichlet data at the root down the tree until all leaf-level nodes have Dirichlet data. This constitutes the family traversal (found in Algorithm 6) that is done in a pre-order fashion.

The leaf traversal is then a final step to iterate over all the leaf-level patches and solve 5.1 on the subdomains using fast methods. This is an independent step for each leaf-level patch and can be done simultaneously everywhere, taking advantage of the domain decomposition.

Algorithm 6 `Split1To4` Function (Solve Stage Family Callback)

Require: $\mathbf{S}^\tau$, $\mathbf{g}^\tau = \mathbf{u}^\tau_{ext}$
  `tag` = `TagForUncoarsening`(τ)
  if `tag` **then**
    Uncoarsen: $\mathbf{g}^\tau = \mathbf{L}_{1,2}\mathbf{g}^\tau$
  Compute $\mathbf{u}^\tau_{\text{int}} = \mathbf{S}^\tau\mathbf{g}^\tau + \mathbf{w}^\tau$
  Partition $\mathbf{u}^\tau_{\text{int}}$ into $\mathbf{u}^\alpha_{\text{ext}}, \mathbf{u}^\beta_{\text{ext}}, \mathbf{u}^\gamma_{\text{ext}}, \mathbf{u}^\omega_{\text{ext}}$

5.3 The Parallel Algorithm

In this section, we detail the data structures and algorithms used in the parallel implementation of the quadtree-adaptive HPS method. We start by going over the leaf-indexed quadtree provided from `p4est` and the path-indexed quadtree. This leads into how these data structures are implemented in parallel, including communication patterns. Finally, this leads to a discussion of the merging and splitting algorithms in parallel.

5.3.1 MPI Preliminaries

We define and assume the following with regard to using the Message Passing Interface (MPI) Message Passing Interface Forum (2021). A *rank* is a compute unit with a separate, unique memory space. Each rank in the execution of a program runs a separate version of the program. *Communication* is how data is shared via *messages* between ranks and is associated with actions such as sending, receiving, or broadcasting messages. A *communicator* is a union of ranks in which each rank has a unique index $R_i = 0, ..., N_R - 1$, where N_R is the number of ranks in a communicator. In every MPI program, there is a global communicator called `MPI_COMM_WORLD` that contains all ranks that are executing the program.

5.3.2 Quadtree Data Structures

The `p4est` library provides a leaf-indexed quadtree data structure and functions to construct, store, and iterate over leaf-level nodes (also called quadrants). The `p4est` quadtree only stores leaf-level quadrants. Quadrants in a `p4est` quadtree are stored locally to each rank, with minimal redundant metadata stored on each rank to handle the logic for where each quadrant is stored. The quadrants are stored in a rank-local linked list and indexed according to a space-filling curve. Partitioning in `p4est` results in an equally-divided quadtree where each rank has roughly the same number of quadrants, subject to the integer division of the number of leaf-nodes N_L into N_R.

As outlined in Chipman *et al.* (2024), the HPS method requires storage for all nodes in a quadtree, including leaves and ancestors. This is to store the set of matrices that act as the factorization of the system matrix. We developed a path-indexed quadtree that builds upon the quadtrees provided in `p4est` and has storage for all nodes in the quadtree. The path-indexed quadtree indexes each node in the tree according to the path of the node, where a path is the unique series of directions required to traverse from the root of the tree to the node in question. Each node can be uniquely indexed according to its path. The path is used as a key in a key-value lookup table (i.e. map) to provide storage for each node in the path-indexed quadtree. Figure 5.2 depicts the leaf-indexed and path-indexed quadtrees, including colors that denote which ranks own each node.

In parallel, the leaf nodes of a leaf-indexed quadtree are partitioned across available ranks. The partitioning of a path-indexed quadtree follows the partitioning of a leaf-indexed quadtree. Any node in a path-indexed quadtree has a range of ranks from $R_{\text{first}}$to R_{last}on which that node exists. A leaf node is always local to a single

rank, or $R_{\text{first}} = R_{\text{last}}$. The parents and ancestors of the leaf nodes can exist across multiple ranks depending on the range of ranks of a node's descendants. The range of ranks is provided from `p4est` upon initialization of the path-indexed quadtree and associated nodes.

The range of ranks allows for creation of a *node communicator*. A node communicator is a subset of the global communicator that includes only the range of ranks a particular node lives on. When communication is necessary (as in the 4-to-1 merge or the 1-to-4 split algorithms), a node may communicate its data to all ranks that store a version of that node in the quadtree. Creation of the node communicator is a collective routine on all ranks that are part of the super communicator. The node communicators are created and stored on each node during the initialization of the path-indexed quadtree to avoid global communication when performing the factorization or solves.

Initialization of a path-indexed quadtree assumes a leaf-indexed quadtree already exists. The leaf-indexed quadtree (a `p4est` object) can be created by refining a quadrant according to a refinement criteria that is problem dependent (see Figure 5.1). The leaf-indexed quadtree is not required to have data associated with each quadrant, so there is minimal redundant logical storage. Once a leaf-indexed quadtree is refined, the path-indexed quadtree can be created through a depth-first traversal of the leaf-indexed quadtree. This utility is provided in the `p4est_search_all` function.

The callback function provided to `p4est_search_all` is found in Algorithm 7. `p4est` also provides a utility to get the index of a quadrant's ancestor in `p4est_quadrant_ancestor_id`. This function is used to create the path of the quadrant (represented as a string) that is used as the key in the map that stores the nodes in the path-indexed quadtree. The

main idea of Algorithm 7 is to allocate space in the map if the rank executing the code owns the node in the quadtree, and if not, set that space in the map to `nullptr`. The node is created either from the root data (provided from the user) or the parent data. In practice, a factory design pattern is used to generate the nodes; the user implements a node factory object.

Algorithm 7 `p4est_search_all_callback` Function

Require: $\mathcal{Q}_L$, $\mathcal{Q}_P$, Ω_i, R_{first}, R_{last}
  Get `map` from $\mathcal{Q}_P$
  Compute `path` from `p4est_quadrant_ancestor_id`(Ω_i)
  Let `owned` = $R_{\text{first}} < r < R_{\text{last}}$
  if `owned` **then**
    if Ω_i is the root quadrant **then**
      Create `node_ptr` from root data: `node_ptr` $\rightarrow \Omega^\tau$
    else
      Create `node_ptr` from parent data: `node_ptr` $\rightarrow \Omega^\tau$
    Let `map[path] = node_ptr`
  else
    Let `map[path] = nullptr`

The merging and splitting algorithms detailed in Section 5.2.3 require references to the data stored on the parent node and the children nodes associated with the merge or split. When children nodes are not rank-local, communication is necessary to share the data across ranks. Ranks within a node communicator broadcast their associated data to other involved ranks to ensure that each rank has a local copy of all children and parent nodes. For example, from Figure 5.2(b), when merging children nodes “0210”, “0211”, “0212”, and “0213” into parent node “021”, the yellow rank must broadcast nodes “0210”, “0211”, and “0212” to the green rank, and the green rank must broadcast node “0213” to the yellow rank. Both yellow and green ranks would already have storage for the parent node “021”.

The algorithm for communicating families of nodes among ranks is outlined in Algorithm 8. This is the function that is passed to `p4est_search_reorder` either for the pre-quadrant callback (splitting algorithms) or the post-quadrant callback (merging algorithms). The primary logic of Algorithm 8 checks if the node is a leaf or not to call either `LeafCallback` or `FamilyCallback` and if the node is local to the rank executing this code. In order to call `MPI_Bcast`, the root rank must be known. This is the rank that sends the data in the broadcast; all other nodes receive the data and store it in a rank-local buffer. A call to `MPI_Allreduce` for each children node communicates which rank is the root rank for that node. The call to `MPI_Bcast` provides the buffer for ranks to send and receive node data and only communicates within the node communicator.

Algorithm 8 `p4est_search_merge_split` Function

Require: $\mathcal{Q}_L$, $\mathcal{Q}_P$, Ω_i, R_{first}, R_{last}
  Get `map` from $\mathcal{Q}_P$
  Compute `path` from `p4est_quadrant_ancestor_id`(Ω_i)
  Let `node_ptr` = `map[path]`: `node_ptr` $\rightarrow \Omega^\tau$
  Let `owned` = $R_{\text{first}} < r < R_{\text{last}}$
  Let `children_ptrs` = {}
  if $i \geq 0$ & `node_ptr` $\neq$ `nullptr` & `owned` **then**
    Call `LeafCallback(node_ptr)`
  else
    if $R_{\text{first}} = R_{\text{last}}$ **then**
      for $i = 0, ..., 3$ **do**
        Let `children_ptrs[i] = map[path + string(i)]`
    else if `owned` **then**
      for $i = 0, ..., 3$ **do**
        Let `child_ptr = map[path + string(i)]`
        Communicate `child_ptr` to all ranks in `node_comm`
        Let `children_ptrs[i] = child_ptr`
    if `owned` **then**
      Call `FamilyCallback(node_ptr, children_ptrs)`

5.4 Parallel Results and Discussion

We demonstrate the strong and weak scaling of the current implementation. For each of the strong and weak scaling runs, we time the leaf and family callback stages for the build, upwards, and solve stages of the HPS method. This provides insight into the scalability of the leaf traversals as well as the family traversals for each stage.

For both strong and weak scaling analysis, we ran on the petascale machine Polaris at Argonne National Laboratory. Polaris is a 560 node machine, with each node consisting of one 2.8GHz AMD EPYC Milan 7543P 32 core CPU with 512 GB of DDR4 RAM.

5.4.1 Strong Scaling

Strong scaling analysis is done by solving a relatively large problem while increasing the number of compute units used to solve the problem. As one increases the parallelism, the compute per compute unit decreases. Ideal strong scaling should result in the time to solution decreasing at the same rate that the number of compute units is increasing.

For this strong scaling analysis, we solve the Poisson equation on an adaptive mesh refined according to the right-hand side. Using the method of manufactured solutions, we generate a Poisson problem based on the exact solution

$$u_{\text{exact}}(x, y) = \sin(x) + \sin(y). \tag{5.17}$$

The resulting Poisson equation is thus

$$\nabla^2 u(x, y) = f(x, y) = -u_{\text{exact}}(x, y), \tag{5.18}$$

where the boundary conditions are provided by evaluating the exact solution on the boundaries. The leaf-indexed and path-indexed quadtrees that represent the underlying mesh are generated by recursively refining the root-level patch according to a refinement criteria of $|f(x, y)| > 1.2$. We used 8 levels of refinement, and we used patch sizes of $M = [8, 16, 32, 64]$. This results in solving the Poisson equation with an effective resolution of up to 8192×8192.

Results and Discussion The results of running on Polaris can be found in Figure 5.3. For each patch size M (indicated by the colors in Figure 5.3), we increase the number of MPI ranks used to solve the problem. While we were able to run on up to 4096 MPI ranks, the communication overhead at larger runs greatly outweighs the amount of compute for this problem. This results in strong scaling that drops off. We show scaling on up to $N_R = 128$ MPI ranks.

Figure 5.3 shows that the leaf callback functions for all three stages scale nearly perfectly. This is to be expected as there is little to no communication at this stage; each partition builds up the factorization or solves the elliptic problem independently. The family callbacks for all three stages, however, have less than optimal scaling. In each stage, the family callback includes communicating data across families. Sibling nodes at intermediate levels (i.e., non-leaf nodes) are often owned by multiple ranks. The communication among ranks at the upper levels involves potentially global communication, including broadcast operations. This amount of communication is what leads to poor strong scaling at medium to larger runs; the amount of communication grows much faster than the amount of compute.

Comparing the strong scaling among stages shows that the solve stage scales better than the build and the upwards stages. The size of the data involved in the

communication in the solve stage is smaller than that in the factorization stages.

5.4.2 Weak Scaling

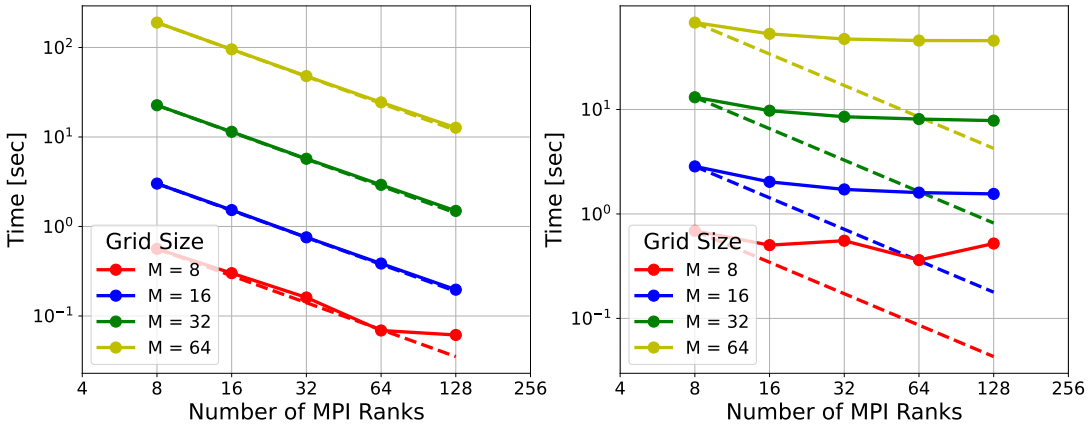
Weak scaling analysis is done by solving a problem while increasing both the size of the problem and the number of compute units. The amount of compute per compute unit stays constant in a weak scaling analysis. The time to solution should stay constant as one increases the parallelism.

For this weak scaling analysis, we solve the "polar star" Poisson problem with one polar star per compute unit. The exact solution we attempt to reconstruct is the following:

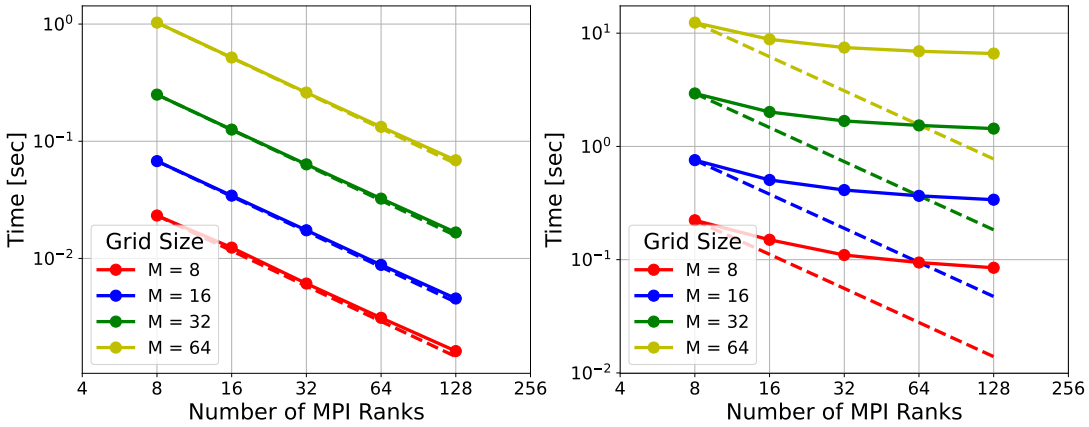
$$u(x,y) = \frac{1}{2}\sum_{i=1}^{N_S} 1 - \tanh\left(\frac{r(x,y) - r_{0,i}\left(r_{1,i}\cos\left(n\theta(x,y)\right)+1\right)}{\epsilon}\right), \tag{5.19}$$

where N_S is the number of polar stars. Computing the Laplacian analytically yields the right-hand side to Poisson's equation. Thus, the polar star Poisson problem is defined as follows:

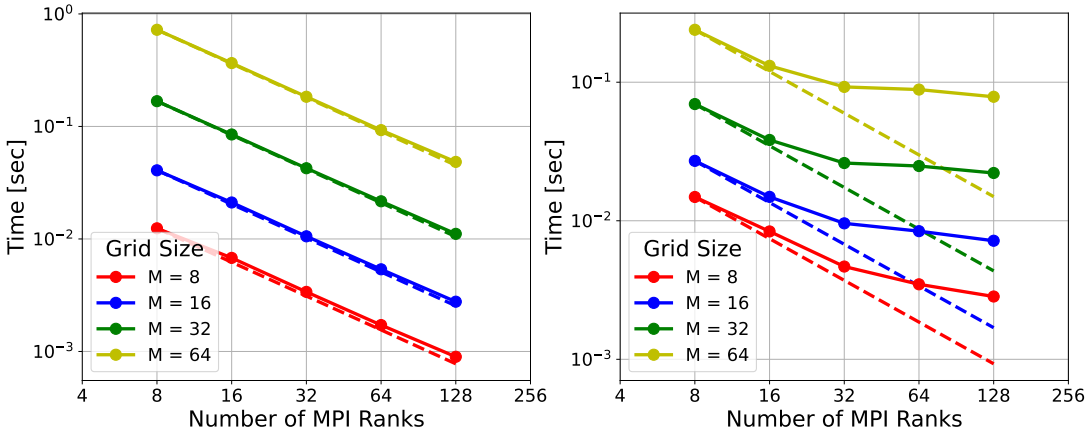
$$\nabla^2 u(x,y) = \sum_{i=1}^{N_S} -\frac{s_{1,i}(x,y) + s_{2,i}(x,y)}{r(x,y)^2} - s_{3,i}(x,y) + s_{4,i}(x,y) \tag{5.20}$$



(a) The build stage strong scaling.



(b) The upwards stage strong scaling.



(c) The solve stage strong scaling.

Figure 5.3: The strong scaling plots for the (a) build stage, (b) upwards stage, and (c) solve stage. For each, the left plot shows the scaling for the leaf callback and the right plot shows the scaling for the family callback. The solid line indicates actual timing and the dashed line indicates ideal strong scaling.

with

$$
\begin{aligned}
s_{1,i}(x,y) &= \frac{p(x,y)^2 \tanh\left(\phi(x,y)\right) \operatorname{sech}^2\left(\phi(x,y)\right)}{\epsilon^2} \\
s_{2,i}(x,y) &= -\frac{n^2 r_{0,i} r_{1,i} \cos(n\theta(x,y)) \operatorname{sech}^2\left(\phi(x,y)\right)}{2\epsilon} \\
s_{3,i}(x,y) &= \frac{\tanh\left(\phi(x,y)\right) \operatorname{sech}^2\left(\phi(x,y)\right)}{\epsilon^2} \\
s_{4,i}(x,y) &= \frac{\operatorname{sech}^2\left(\phi(x,y)\right)}{2r(x,y)\epsilon} \\
p(x,y) &= n r_{0,i} r_{1,i} \sin(n\theta(x,y)) \\
\phi(x,y) &= \frac{r(x,y) - r_{0,i}(r_{1,i}\cos(n\theta(x,y)) + 1)}{\epsilon}.
\end{aligned}
$$

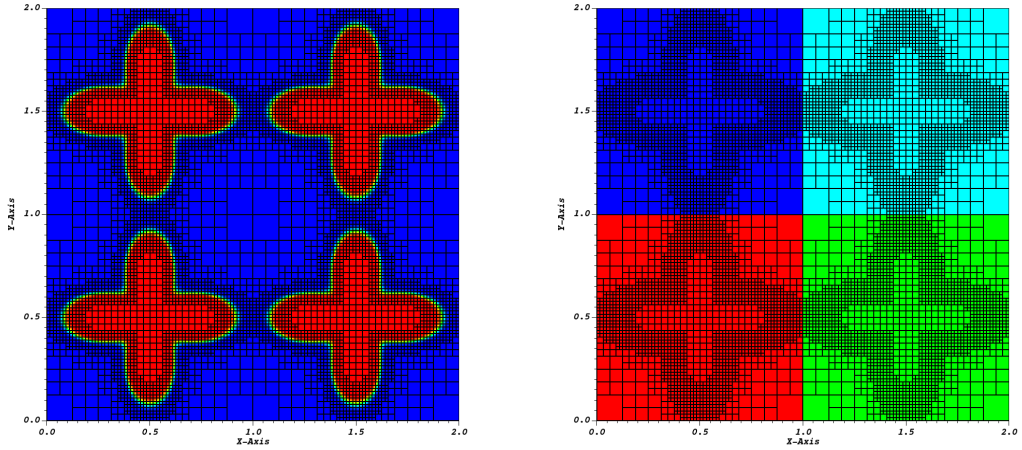
Each polar star has a center (x_0, y_0), inner and outer radii r_0, r_1, and the number of arms per polar star n. The radius and angle have the standard polar transforms:

$$r(x,y) = \sqrt{(x-x_0)^2 + (y-y_0)^2} \tag{5.21}$$

$$\theta(x,y) = \tan^{-1}\left(\frac{y-y_0}{x-x_0}\right) \tag{5.22}$$

To keep the number of degrees of freedom constant per MPI rank, we solve on a square domain with a single polar star at the center. With each increase in the number of MPI ranks, we extend the domain to include more polar stars, each at the center of their own partition.

For this study, we refine around the edges of the polar stars. The refinement criteria was $|f(x,y)| > 1$, with 7 levels of refinement. Each polar star was a 4-pointed polar star with $r_0 = 0.3$, $r_1 = 0.4$, and $\epsilon = 0.015625$. We solve with patch sizes $M = [16, 32, 64]$, resulting in the degrees of freedom per MPI rank to be $[4096, 16384, 65536]$, respectively.



(a) The plotted solution. **(b) The MPI partitions.**

Figure 5.4: Plots of the polar star Poisson problem used in the weak scaling study found in Section 5.4.2. The polar star is generated from the RHS of the Poisson equation and is repeated for each MPI rank used to solve the problem. This shows four polar stars arranged in a 2×2 grid for solving with $N_R = 4$. Each outlined grid contains a 16×16 finite volume mesh.

An example of this setup can be found in Figure 5.4, which show the repeated solution and partitioning for $N_R = 4$ arranged in a 2×2 grid of processes.

Results and Discussion Figure 5.5 contains the results of the weak scaling runs. We plot the weak scaling efficiency, which is computed as

$$E_{\text{weak}} = \frac{t_1}{t_{N_R}} \times 100\%, \tag{5.23}$$

where t_1 is the amount of time spent in that stage on a single MPI rank. The colors indicate the different patch sizes, and the leaf and family callbacks are shown for all three stages.

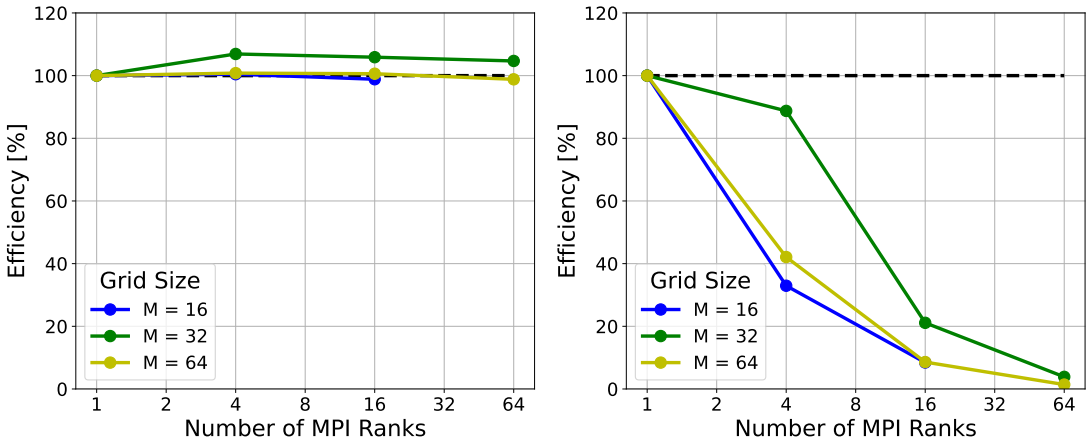
Similar to the strong scaling, the leaf callbacks scale better than the family callbacks. The build stage and solve stage leaf callbacks maintain nearly perfect weak scaling

efficiency. The upwards stage scales similarly to the scaling of the family callbacks, which is unexpected. The family callbacks demonstrate similar scaling drop off as in the strong scaling. Again, this is likely due to the increased communication, along with the redundant calculations exhibited in this implementation.

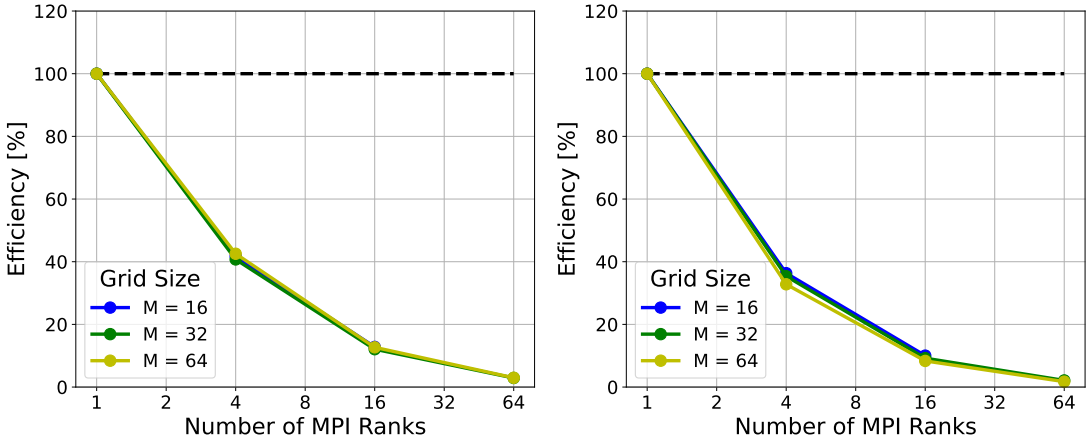
5.5 Conclusion

The work presented in this paper demonstrates a successful implementation of the quadtree-adaptive Hierarchical Poincaré-Steklov (HPS) method in parallel. The integration of the `p4est` library, known for its efficiency and scalability, with the communication patterns of the quadtree-adaptive HPS method, shows promise for an effective solver for elliptic PDEs. As distributed memory architectures are increasingly more common in commodity clusters, an MPI implementation is essential for future scalability.

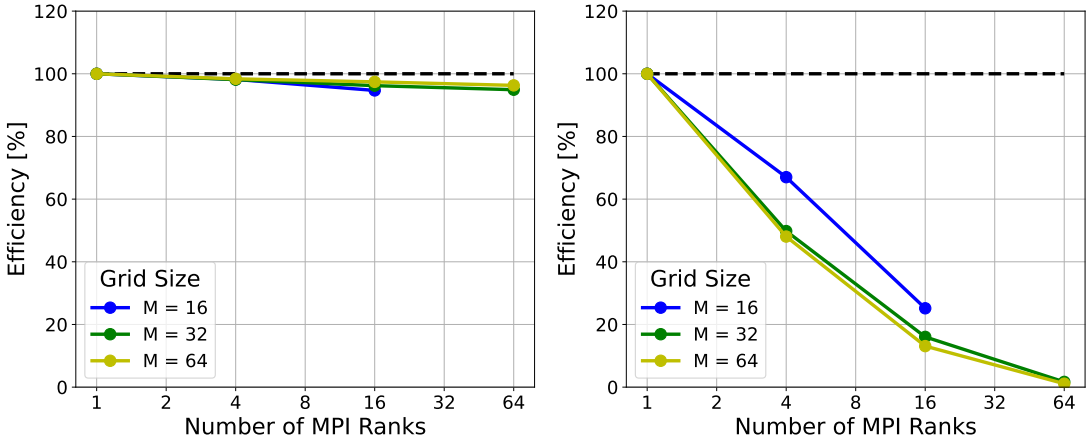
Running strong and weak scaling analysis on Polaris, one of the world's top 50 fastest supercomputers, provides insights into potential and necessary improvements for further optimizations. The current bottleneck is the communication required for the family callbacks during family traversals of the path-indexed quadtree. Potential optimizations include reducing redundant calculations through more efficient matrix partitioning and investigating non-blocking communication to better overlap compute and communication. These insights are currently being investigated and integrated into EllipticForest Chipman (2024).



(a) The build stage weak scaling.



(b) The upwards stage weak scaling.



(c) The solve stage weak scaling.

Figure 5.5: The weak scaling plots for the (a) build stage, (b) upwards stage, and (c) solve stage. For each, the left plot shows the scaling for the leaf callback and the right plot shows the scaling for the family callback. The solid line indicates actual efficiency and the dashed line indicates ideal weak scaling.

CHAPTER 6:

APPLICATIONS OF THE HPS METHOD TO COUPLED SYSTEMS OF ELLIPTIC PDES

CHAPTER 7:

CONCLUSION

REFERENCES

Adams, John C, Swarztrauber, Paul N, & Sweet, Roland. 2016. FISHPACK90: Efficient fortran subprograms for the solution of separable elliptic partial differential equations. *Astrophysics Source Code Library*, ascl–1609.

Aiton, Scott, Calhoun, Donna, & Wright, Grady. 2022. *ThunderEgg*. url=https://github.com/ThunderEgg/ThunderEgg.

Anderson, Edward, Bai, Zhaojun, Bischof, Christian, Blackford, L Susan, Demmel, James, Dongarra, Jack, Du Croz, Jeremy, Greenbaum, Anne, Hammarling, Sven, McKenney, Alan, *et al.* 1999. *LAPACK users' guide*. SIAM.

Babb, Tracy, Gillman, Adrianna, Hao, Sijia, & Martinsson, Per-Gunnar. 2018. An accelerated Poisson solver based on multidomain spectral discretization. *BIT Numerical Mathematics*, **58**(4), 851–879.

Balay, Satish, Abhyankar, Shrirang, Adams, Mark F., Benson, Steven, Brown, Jed, Brune, Peter, Buschelman, Kris, Constantinescu, Emil, Dalcin, Lisandro, Dener, Alp, Eijkhout, Victor, Faibussowitsch, Jacob, Gropp, William D., Hapla, Václav, Isaac, Tobin, Jolivet, Pierre, Karpeev, Dmitry, Kaushik, Dinesh, Knepley, Matthew G., Kong, Fande, Kruger, Scott, May, Dave A., McInnes, Lois Curfman, Mills, Richard Tran, Mitchell, Lawrence, Munson, Todd, Roman, Jose E., Rupp, Karl, Sanan, Patrick, Sarich, Jason, Smith, Barry F., Zampini, Stefano, Zhang,

Hong, Zhang, Hong, & Zhang, Junchao. 2023. *PETSc/TAO Users Manual*. Tech. rept. ANL-21/39 - Revision 3.20. Argonne National Laboratory.

Beams, Natalie N, Gillman, Adrianna, & Hewett, Russell J. 2020. A parallel shared-memory implementation of a high-order accurate solution technique for variable coefficient Helmholtz problems. **79**(4), 996–1011.

Brandt, Achi. 1977. Multi-level adaptive solutions to boundary-value problems. **31**(138), 333–390.

Briggs, William L, Henson, Van Emden, & McCormick, Steve F. 2000. *A multigrid tutorial*. SIAM.

Burstedde, Carsten. 2020. Parallel tree algorithms for AMR and non-standard data access. *ACM Transactions on Mathematical Software (TOMS)*, **46**(4), 1–31.

Burstedde, Carsten, Wilcox, Lucas C., & Ghattas, Omar. 2011. `p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, **33**(3), 1103–1133.

Calhoun, Donna, & Burstedde, Carsten. 2017. ForestClaw: A parallel algorithm for patch-based adaptive mesh refinement on a forest of quadtrees. *arXiv preprint arXiv:1703.03116*.

Cheng, Hongwei, Huang, Jingfang, & Leiterman, Terry Jo. 2006. An adaptive fast solver for the modified Helmholtz equation in two dimensions. *Journal of Computational Physics*, **211**(2), 616–637.

Childs, Hank, Brugger, Eric, Whitlock, Brad, Meredith, Jeremy, Ahern, Sean, Pugmire, David, Biagas, Kathleen, Miller, Mark, Harrison, Cyrus, Weber, Gunther H.,

Krishnan, Hari, Fogal, Thomas, Sanderson, Allen, Garth, Christoph, Bethel, E. Wes, Camp, David, Rübel, Oliver, Durant, Marc, Favre, Jean M., & Navrátil, Paul. 2012. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. *Pages 357–372 of: High Performance Visualization–Enabling Extreme-Scale Scientific Insight*.

Chipman, Damyn. 2023 (January). *EllipticForest*. Website.

Chipman, Damyn. 2024. EllipticForest: A Direct Solver Library for Elliptic Partial Differential Equations on Adaptive Meshes. *Journal of Open Source Software*, **9**(96), 6339.

Chipman, Damyn, Calhoun, Donna, & Burstedde, Carsten. 2024. *A Fast Direct Solver for Elliptic PDEs on a Hierarchy of Adaptively Refined Quadtrees*.

Chow, Edmond, Falgout, Robert D, Hu, Jonathan J, Tuminaro, Raymond S, & Yang, Ulrike Meier. 2006. A survey of parallelization techniques for multigrid solvers. *Parallel processing for scientific computing*, 179–201.

Colella, Phillip, Graves, Daniel T, Ligocki, TJ, Martin, DF, Modiano, D, Serafini, DB, & Van Straalen, B. 2009. Chombo software package for AMR applications design document. *Available at the Chombo website: http://seesar. lbl. gov/ANAG/chombo/(September 2008)*.

Davis, Timothy A. 2004. Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software (TOMS)*, **30**(2), 196–199.

Demmel, James W, Gilbert, John R, & Li, Xiaoye S. 1999. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, **20**(4), 915–952.

Donfack, Simplice, Dongarra, Jack, Faverge, Mathieu, Gates, Mark, Kurzak, Jakub, Luszczek, Piotr, & Yamazaki, Ichitaro. 2015. A survey of recent developments in parallel implementations of Gaussian elimination. *Concurrency and Computation: Practice and Experience*, **27**(5), 1292–1309.

Dubey, Anshu, Weide, Klaus, O'Neal, Jared, Dhruv, Akash, Couch, Sean, Harris, J Austin, Klosterman, Tom, Jain, Rajeev, Rudi, Johann, Messer, Bronson, *et al.* 2022. Flash-X: A multiphysics simulation software instrument. *SoftwareX*, **19**, 101168.

Falgout, Robert D, & Yang, Ulrike Meier. 2002. hypre: A library of high performance preconditioners. *Pages 632–641 of: International Conference on computational science*. Springer.

Fortunato, Daniel, Hale, Nicholas, & Townsend, Alex. 2020. The ultraspherical spectral element method. 110087.

Geldermans, Peter, & Gillman, Adrianna. 2019. An adaptive high order direct solution technique for elliptic boundary value problems. **41**(1), A292–A315.

George, Alan. 1973. Nested dissection of a regular finite element mesh. **10**(2), 345–363.

Gillman, Adrianna, & Martinsson, Per-Gunnar. 2014. A direct solver with O(N) complexity for variable coefficient elliptic PDEs discretized via a high-order com-

posite spectral collocation method. *SIAM Journal on Scientific Computing*, **36**(4), A2023–A2046.

Globisch, Gerhard. 1995. PARMESH—a parallel mesh generator. *Parallel Computing*, **21**(3), 509–524.

Glusa, Christian, Boman, Erik G, Chow, Edmond, Rajamanickam, Sivasankaran, & Szyld, Daniel B. 2020. Scalable asynchronous domain decomposition solvers. *SIAM Journal on Scientific Computing*, **42**(6), C384–C409.

Gupta, Anshul, Karypis, George, & Kumar, Vipin. 1997. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed systems*, **8**(5), 502–520.

Hao, Sijia, & Martinsson, Per-Gunnar. 2016. A direct solver for elliptic PDEs in three dimensions based on hierarchical merging of Poincaré–Steklov operators. *Journal of Computational and Applied Mathematics*, **308**, 419–434.

Hestenes, Magnus Rudolph, Stiefel, Eduard, *et al.* 1952. *Methods of conjugate gradients for solving linear systems*. Vol. 49. NBS Washington, DC.

Higham, Nicholas J. 2002. *Accuracy and stability of numerical algorithms*. SIAM.

Hornung, Richard D, Wissink, Andrew M, & Kohn, Scott R. 2006. Managing complex data and geometry in parallel structured AMR applications. *Engineering with Computers*, **22**, 181–195.

Kagan, Pavel, Fischer, Anath, & Bar-Yoseph, Pinhas Z. 1998. New B-Spline Finite Element approach for geometrical design and mechanical analysis. **41**(3), 435–458.

LeVeque, Randall J. 2007. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*. SIAM.

Li, Xiaoye S. 2005. An Overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Transactions on Mathematical Software*, **31**(3), 302–325.

Lipton, Richard J, Rose, Donald J, & Tarjan, Robert Endre. 1979. Generalized nested dissection. **16**(2), 346–358.

Liu, Yang. 2018. *ButterflyPACK*. Tech. rept. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

Martinsson, Per-Gunnar. 2019. *Fast direct solvers for elliptic PDEs*. SIAM.

Martinsson, P.G. 2013. A direct solver for variable coefficient elliptic PDEs discretized via a composite spectral collocation method. **242**, 460–479.

Martinsson, PG. 2015. The hierarchical Poincaré-Steklov (HPS) solver for elliptic PDEs: A tutorial. *arXiv preprint arXiv:1506.01308*.

Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard Version 4.0*.

Ortega, James M, & Voigt, Robert G. 1985. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, **27**(2), 149–240.

P. Martinsson, V. Rokhlin. 2004. A fast direct solver for boundary integral equations in two dimensions.

Patera, Anthony T. 1984. A spectral element method for fluid dynamics: laminar flow in a channel expansion. **54**(3), 468–488.

Quarteroni, Alfio, & Valli, Alberto. 1991. Theory and application of Steklov-Poincaré operators for boundary-value problems. *Pages 179–203 of: Applied and Industrial Mathematics*. Springer.

Quarteroni, Alfio, Sacco, Riccardo, & Saleri, Fausto. 2010. *Numerical mathematics*. Vol. 37. Springer Science & Business Media.

Saad, Youcef, & Schultz, Martin H. 1986. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. **7**(3), 856–869.

Samet, Hanan. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, **16**(2), 187–260.

Smith, Barry F. 1997. Domain decomposition methods for partial differential equations. *Pages 225–243 of: Parallel Numerical Algorithms*. Springer.

Sundar, Hari, Biros, George, Burstedde, Carsten, Rudi, Johann, Ghattas, Omar, & Stadler, Georg. 2012. Parallel geometric-algebraic multigrid on unstructured forests of octrees. *Pages 1–11 of: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE.

Swarztrauber, P, Sweet, R, & Adams, John C. 1999. FISHPACK: Efficient FORTRAN subprograms for the solution of elliptic partial differential equations. *UCAR Publication, July*.

Townsend, Alex, & Olver, Sheehan. 2015. The automatic solution of partial differential equations using a global spectral method. **299**, 106–123.

Trefethen, Lloyd N, & Bau III, David. 1997. *Numerical linear algebra*. Vol. 50. SIAM.

Woodward, JR. 1982. The explicit quad tree as a structure for computer graphics. *The Computer Journal*, **25**(2), 235–238.

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

Yesypenko, Anna, & Martinsson, Per-Gunnar. 2022a. GPU Optimizations for the Hierarchical Poincaré-Steklov Scheme. *Pages 519–528 of: International Conference on Domain Decomposition Methods*. Springer.

Yesypenko, Anna, & Martinsson, Per-Gunnar. 2022b. Parallel Optimizations for the Hierarchical Poincar\'e-Steklov Scheme (HPS). *arXiv preprint arXiv:2211.14969*.

Zhang, Weiqun, Almgren, Ann, Beckner, Vince, Bell, John, Blaschke, Johannes, Chan, Cy, Day, Marcus, Friesen, Brian, Gott, Kevin, Graves, Daniel, *et al.* 2019. AMReX: a framework for block-structured adaptive mesh refinement. *The Journal of Open Source Software*, **4**(37), 1370.

APPENDIX A:

RELEVANT SOFTWARE

A.1 EllipticForest

Should I reference JOSS here?

A.1.1 Summary

EllipticForest is a software library with utilities to solve elliptic partial differential equations (PDEs) with adaptive mesh refinement (AMR) using a direct matrix factorization. It implements a quadtree-adaptive variation of the Hierarchical Poincaré-Steklov (HPS) method Gillman & Martinsson (2014). The HPS method is a direct method for solving elliptic PDEs based on the recursive merging of Poincaré-Steklov operators Quarteroni & Valli (1991). EllipticForest is built on top of the parallel and highly efficient mesh library ‘p4est‘ Burstedde *et al.* (2011) for mesh adaptivity and mesh management. Distributed memory parallelism is implemented through the Message Passing Interface (MPI) **?**. EllipticForest wraps the fast, cyclic-reduction methods found in the FISHPACK Swarztrauber *et al.* (1999) library and updated in the FISHPACK90 Adams *et al.* (2016) library at the lowest grid level (called leaf patches). In addition, for more general elliptic problems, EllipticForest wraps solvers from the PDE solver library PETSc Balay *et al.* (2023). The numerical methods used in EllipticForest are detailed in Chipman *et al.* (2024). A key feature of EllipticForest is the ability

for users to extend the solver interface classes to implement custom solvers on leaf patches. EllipticForest is an implementation of the HPS method to be used as a software library, either as a standalone to solve elliptic PDEs or for coupling with other scientific libraries for broader applications.

A.1.2 Statement of Need

Elliptic PDEs arise in a wide-range of physics and engineering applications, including fluid modeling, electromagnetism, astrophysics, heat transfer, and more. Solving elliptic PDEs is often one of the most computationally expensive steps in numerical algorithms due to the need to solve large systems of equations. Parallel algorithms are desirable in order solve larger systems at scale on small to large computing clusters. Communication patterns for elliptic solvers make implementing parallel solvers difficult due to to the global nature of the underlying mathematics. In addition, adaptive mesh refinement adds coarse-fine interfaces and more complex meshes that make development and scalability difficult. The solvers implemented in EllipticForest address these complexities through proven numerical methods and efficient software implementations.

The general form of elliptic PDEs that EllipticForest is tailored to solve is the following:

$$\alpha(x,y)\nabla\cdot\Big[\beta(x,y)\nabla u(x,y)\Big]+\lambda(x,y)u(x,y)=f(x,y) \tag{A.1}$$

where $\alpha(x,y)$, $\beta(x,y)$, $\lambda(x,y)$, and $f(x,y)$ are known functions in x and y and the goal is to solve for $u(x,y)$. Currently, EllipticForest solves the above problem in

a rectangular domain $\Omega = [x_L, x_U] \times [y_L, y_U]$. The above PDE is discretized using a finite-volume approach using a standard five-point stencil yielding a second-order accurate solution. This leads to a standard linear system of equations of the form

$$\mathbf{A}\mathbf{u} = \mathbf{f} \tag{A.2}$$

which is solved via the HPS method, a direct matrix factorization method.

Similar to other direct methods, the HPS method is comprised of two stages: a build stage and a solve stage. In the build stage, a set of solution operators are formed that act as the factorization of the system matrix corresponding to the discretization stencil. This is done with $\mathcal{O}(N^{3/2})$ complexity, where N is the size of the system matrix. In the solve stage, the factorization is applied to boundary and non-homogeneous data to solve for the solution vector with linear complexity $\mathcal{O}(N)$. The build and the solve stages are recursive applications of a merge and a split algorithm, respectively. The advantages of this approach over iterative methods such as conjugate gradient and multi-grid methods include the ability to apply the factorization to multiple right-hand side vectors.

In addition, another advantage of the quadtree-adaptive HPS method as implemented in EllipticForest is the ability to adapt the matrix factorization to a changing grid. When the mesh changes due to a refining/coarsening criteria, traditional matrix factorizations must be recomputed. The quadtree-adaptive HPS method can adapt the factorization locally, eliminating the need to recompute the factorization. This works as the HPS method builds a set of solution operators that act like a global solution operator. When the mesh changes, the set can be updated by locally ap-

plying the merging and splitting algorithms. This is especially practical for implicit time-dependent problems that require a full linear solve each time step.

EllipticForest builds upon the ‘p4est‘ mesh library Burstedde *et al.* (2011). The quadtree-adaptive HPS method is uniquely suited for quadtree meshes. ‘p4est‘, as a parallel and highly efficient mesh library, provides routines for creating, adapting, and iterating over quadtree meshes. The routines in EllipticForest wrap or extend the capabilities in ‘p4est‘. A primary extension is the development of a *path-indexed* quadtree. This is in contrast to the *leaf-indexed* quadtree implemented in ‘p4est‘. A *path-indexed* quadtree is a data structure that stores data at all nodes in a quadtree, as opposed to just the leaf nodes (see Figure A.1). The *path-indexed* quadtree data structure is designed to store the various data and operators required in the quadtree-adaptive HPS method.

The novelty of EllipticForest as software is the implementation of the HPS method for coupling with other scientific software as well as user extension. Currently, other implementations of the HPS method are MATLAB or Python codes designed by research groups and used in-house for solving specific problems **???**. EllipticForest is designed to be extended and coupled with external libraries. This paper highlights the software details including the user-friendly interface to the HPS method and the ability for users to extend the solver interface using modern object-oriented programming (OOP) paradigms.

A.1.3 Software Overview

Below, we outline various components of the software implemented in EllipticForest. These classes and utilities allow the user to create and refine meshes tailored for their use case, initialize the solver for the elliptic PDE, and visualize the output solution.

A user may also extend the functionality of EllipticForest through inheritance of the ‘Patch‘ classes for user-defined solvers at the leaf level.

Quadtree

The underlying data structure that encodes the mesh is a path-indexed quadtree. The ‘Quadtree‘ object is a class that implements a path-indexed quadtree using a ‘NodeMap‘, which is equivalent to ‘std::map¡std::string, Node¡T¿*¿‘. The template parameter ‘T‘ refers to the type of data that is stored on quadtree nodes. The ‘Quadtree‘ implemented in EllipticForest wraps the ‘p4est‘ leaf-indexed quadtree to create, iterate, and operate on the path-indexed quadtree. Functions to iterate over the quadtree include ‘traversePreOrder‘, ‘traversePostOrder‘, ‘merge‘, and ‘split‘. The ‘traversePreOrder‘ and ‘traversePostOrder‘ functions iterate over the tree in a pre- and post-order fashion, respectively, and provide the user with access to the node or node data via a provided callback function. The ‘merge‘ and ‘split‘ functions iterate over the tree in a post- and pre-order fashion, respectively, and provide the user with access to a family of nodes, or a group of four siblings and their parent node.

Mesh

The user interfaces with the domain discretization through the ‘Mesh‘ class. The ‘Mesh‘ class has an instance of the ‘Quadtree‘ detailed above. ‘Mesh‘ provides functions to iterate over patches or cells via ‘iteratePatches‘ or ‘iterateCells‘.

‘Mesh‘ also provides the user with an interface to the visualization features of EllipticForest. A user may add mesh functions via ‘addMeshFunction‘, which are functions in x and y that are defined over the entire mesh. This can either be a

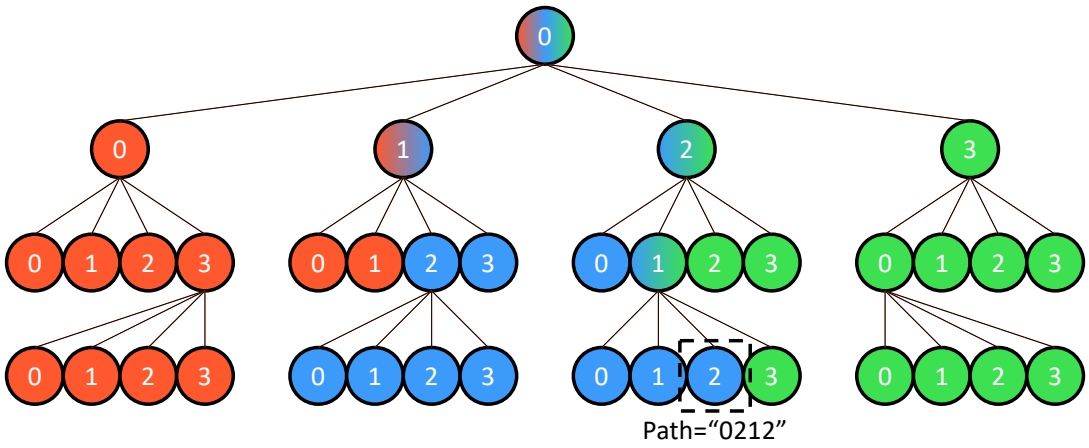


Figure A.1: A path-indexed quadtree representation of a mesh. Colors indicate which rank owns that node. The nodes colored by gradient indicate they are owned by multiple ranks.

mathematical function $f(x, y)$ that is provided via a 'std::function¡double(double x, double y)¿', or as a 'Vector¡double¿' that has the value of $f(x, y)$ at each cell in the domain, ordered by patch and then by the ordering of patch grid. Once a mesh function is added to the mesh, the user may call 'toVTK', which writes the mesh to a parallel, unstructured VTK file format. See the section below on output and visualization for more information.

Patches

The fundamental building block of the mesh and quadtree structures are the patches. A 'Patch' is a class that contains data matrices and vectors that store the solution data and operators needed in the HPS method. A 'Patch' also has an instance of a 'PatchGrid' which represents the discretization of the problem. Each node in the

path-indexed quadtree stores a pointer to a ‘Patch‘.

In EllipticForest, the patch, patch grid, patch solver, and patch node factory interfaces are implemented as a pure virtual interface for the user to extend. Internally, EllipticForest uses these interfaces to call the implemented solver or discretization. By default, EllipticForest implements a 2nd-order, finite volume discretization and solver. This implementation is found under ‘src/Patches/FiniteVolume‘ and each class therein implements the pure virtual interface of the ‘Patch‘, ‘PatchGrid‘, ‘PatchSolver‘, and ‘AbstractNodeFactory‘ classes. Users may use the finite volume implementation shipped with EllipticForest, or they may implement different solvers to be used in the HPS method.

HPS Solver

Once the mesh has been created and refined, and the patch solver has been initialized, solving the elliptic problem on the input mesh is done by creating an instance of the ‘HPSAlgorithm‘ class. The ‘HPSAlgorithm‘ class has member functions that perform the setup, build, upwards, and solve stages of the HPS method. As the HPS method is a direct method, once the build stage has been completed, the upwards and solve stages can be called without rebuilding the matrix factorization.

Output and Visualization

Once the problem has been solved over the entire mesh, each leaf patch in the mesh has the solution stored in one of its data vectors, ‘vectorU‘. This is a discrete representation of the solution to the PDE.

The user may choose to output the mesh and solution in an unstructured PVTK

format using the VTK functionality built-in. To output to VTK files, the user first adds mesh functions to the mesh. This includes the solution stored in ‘vectorU‘ after the HPS solve. Then, the user calls the ‘toVTK‘ member function of the ‘Mesh‘ class. This will write a ‘.pvtu‘ file for the mesh and a ‘.pvtu‘ file for the quadtree. An example of this output for a Poisson equation is shown in Figure A.2.

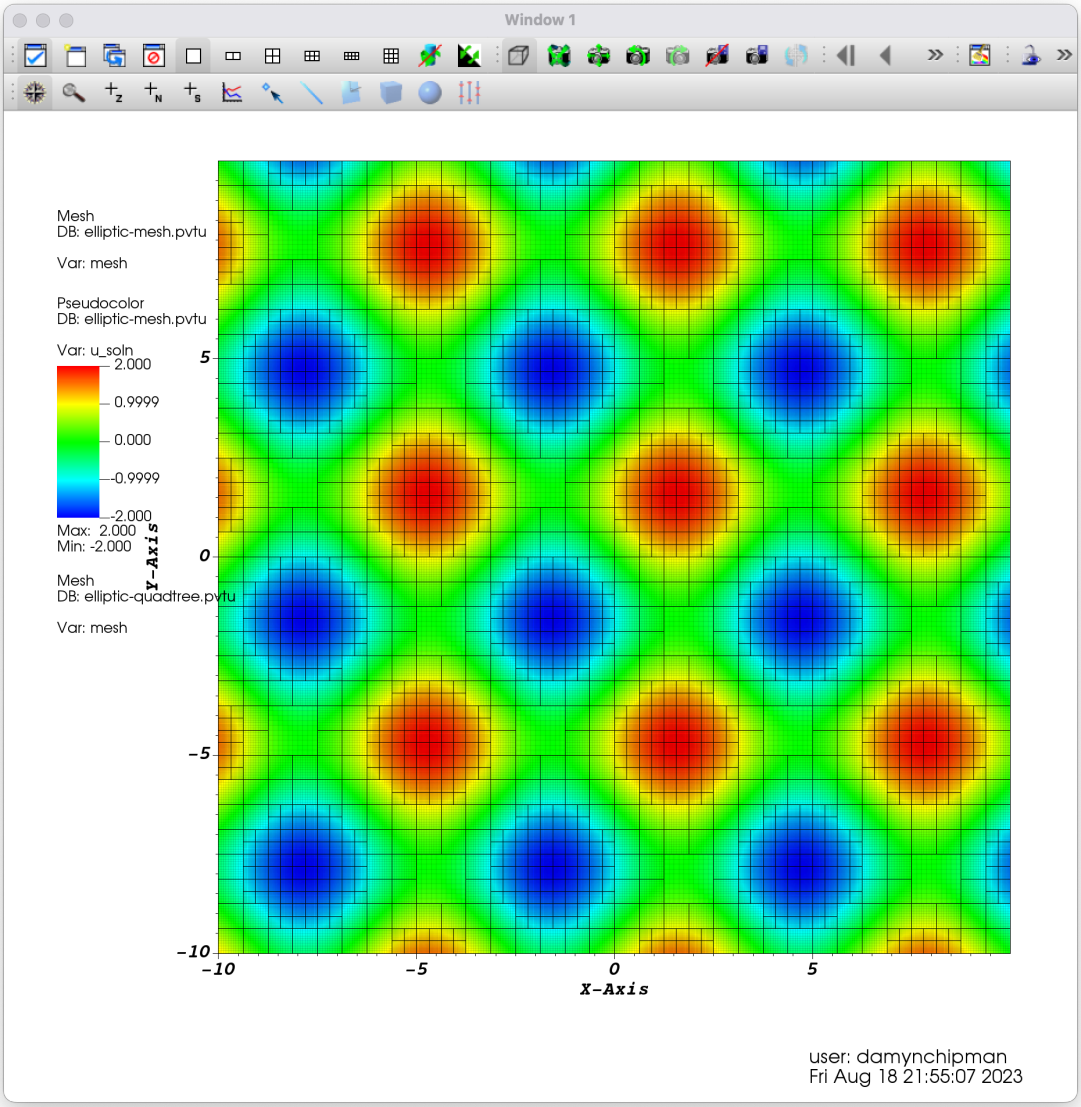


Figure A.2: Solution of Poisson equation on a quadtree mesh using EllipticForest. The mesh and data are output in an unstructured PVTK format and visualized with VisIt ?.