PARTITIONING OPTIMIZATION FOR MASSIVELY PARALLEL TRANSPORT SWEEPS ON UNSTRUCTURED GRIDS

A Dissertation

by

TAREK HABIB GHADDAR

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jean C. Ragusa
Committee Members,	Marvin L. Adams
	Nancy Amato
	Jim E. Morel
Head of Department,	Michael Nastasi

December 2019

Major Subject: Nuclear Engineering

ABSTRACT

The field of radiation transport studies the distribution of radiation throughout a seven-dimensional phase-space consisting of time, space, energy, and direction. Radiation transport is described by the Boltzmann equation that can be solved stochastically or deterministically.

The work presented in this dissertation utilizes the deterministic method known as the transport sweep, a popular technique that has been the subject of a large amount of research. We specifically focus on the parallel implementations of the transport sweep, and predicting the time it takes to sweep across a structured or unstructured mesh given a set of partitioning parameters, achieved through a time-to-solution estimator, written in Python. The time-to-solution estimator is tested against PDT, Texas A&M's massively deterministic transport code. The time-to-solution estimator's sweep time is within 10% of PDT's sweep time for the majority of problems tested.

We use the time-to-solution estimator as the objective function in an optimization scheme to attempt to get the partitions that lead to the fastest sweep time for a given problem and partitioning scheme. Two optimization methods are discussed: using a black box tool (scipy's optimize library) and an intuitive method that prioritizes placing partitions in mesh locations that does not increase the number of cells (which we chose to name the CDF method). The time-to-solution estimator proved to not be smooth enough for a black box tool to work, so the CDF optimization method became the primary method. The CDF method proved effective for the majority of problems run, improving the time to solution over previously used partitioning schemes.

DEDICATION

To my parents, without whom I never would have made it this far.

To Annabelle, who motivated me to keep going when no one else could.

For KHUMNA, who granted me the chaotic energy to persevere through something as crazy as a PhD.

ACKNOWLEDGMENTS

I would like to thank Dr. Jean Ragusa for toeing the line between demanding and understanding perfectly. I know I haven't made your job as my advisor easy.

Thank you to my committee, Drs. Adams, Amato, and Morel, for the feedback and advice when needed to push this work to completion.

A special thank you to Daryl Hawkins, a man I'm convinced is the most overworked code developer on the planet. This work would have been impossible without him.

Thank you to Andrew Till for always being a sounding board for academic ideas, and thank you to Ian Halvic for dealing with my less than fine moments in the office.

Thank you to Dillon Herring for his accelerated effort on his Master's project, which helped the completion of this dissertation.

To Nicolas Quintanar, Brian Ng, and David Saucier, thank you for keeping my life fun and full of life.

To the CERT team, thank you for your constant support in all of my research endeavors.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Jean Ragusa, Marvin Adams, and Jim Morel of the Department of Nuclear Engineering at Texas A&M University, and Professor Nancy Amato of Computer Science at the University of Illinois.

PDT, Texas A&M's massively parallel deterministic transport code is used extensively in this dissertation work. PDT is developed and supported by the Center for Exascale Radiation Transport, (CERT) at Texas A&M.

Chapter 4 was contributed to by Michael Adams and Richard Vega while they were employed or students at Texas A&M University.

The timing constants used in Chapters 5 and 6 were generated as part of Dillon Herring's Master's research at Texas A&M University.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

This material is based upon work supported by the Department of Energy, NationalNuclear Security Administration, under Award Number(s) DE-NA0002376.

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

CONTRIBUTORS AND FUNDING SOURCES v

TABLE OF CONTENTS vi

LIST OF FIGURES viii

LIST OF TABLES xii

1. INTRODUCTION 1

2. THE TRANSPORT SWEEP 4

3. PARALLEL TRANSPORT SWEEPS 7

    3.1 The KBA Algorithm 9
    3.2 PDT's Extension of KBA 12
    3.3 PDT's Performance Model 13

4. UNSTRUCTURED MESHING AND LOAD BALANCING IN PDT 15

    4.1 Original Load-Balancing Algorithm 18
    4.2 Load-Balancing-by-Dimension Algorithm 20
    4.3 Parametric Study of the Original Load Balancing and the Load-Balancing-by-Dimension Algorithms 24
        4.3.1 Parametric study on the unbalanced pin mesh 24
        4.3.2 Parametric study on the Level-2 experiment mesh 26

5. TIME-TO-SOLUTION ESTIMATOR 32

    5.1 Graph Theory Applicable to the Time-to-Solution Estimator 32
    5.2 Method 36
        5.2.1 Building the adjacency matrices 36
        5.2.2 Building the directed acyclic graphs (DAGs) 37
            5.2.2.1 Building the 2D graphs 37

5.2.3 Weighting the task dependence graphs 41
5.2.3.1 Determining the cells per subset for unstructured meshes 43
5.2.4 Adding graphs for angular pipelining 46
5.2.5 Modifying the weights of each graph to reflect a universal timescale 47
5.2.6 Modifying the weights of each graph to reflect conflict resolution 48
5.2.7 Estimating the final time-to-solution 49
5.3 2D Verification 49
5.3.1 Regular partitions 50
5.3.2 "Mildly random" partitions 52
5.3.3 Random partitions 54
5.3.4 Probable worst-case partitions 56
5.4 3D Verification 58
5.5 PDT's Performance Model vs. Time-to-Solution Estimator 59
5.6 PDT vs. Time-to-Solution Estimator for Unstructured Meshes 64
5.6.1 PDT vs. the time-to-solution estimator for the unbalanced pin mesh 64
5.6.2 PDT vs. the time-to-solution estimator for the Level-2 experiment mesh 71

6. PARTITIONING OPTIMIZATION 74

6.1 Scipy Optimize 74
6.2 CDF Optimization 74
6.2.1 Finding the most suitable natural boundaries 76
6.2.2 Finding natural boundaries for sets of columns 78
6.3 Optimization Results 82

7. SUMMARY AND CONCLUSIONS 86

7.1 Future Work 87

REFERENCES 90

APPENDIX A. OPTIMIZATION RESULTS OBTAINED BY SNAPPING CUTS TO THE LARGEST JUMPS 93

LIST OF FIGURES

FIGURE Page

2.1 A demonstration of a sweep on structured and unstructured meshes. 6

3.1 An example showing the pipelining of angular work from the lower left quadrant. ... 10

3.2 An example showing pipelining in the X-Z plane with $A_z = 1$, $P_x = 2$ and $P_z = 1$. . 11

4.1 An unstructured mesh partitioned into 100 subsets with cut lines at 1 cm intervals in both dimensions 16

4.2 A hanging node (circled in black) on a subset boundary (highlighted in blue)..... 17

4.3 The use of the CDF of cells per column to redistribute the cut lines in X. 18

4.4 A 2D subset layout with M unaligned patches of high mesh density N. 20

4.5 A demonstration of the original load balancing and load-balancing-by-dimension algorithms on an unstructured mesh partitioned into 3x3 subsets..... 23

4.6 The results of the parametric study with no load balancing, 5 original load balancing iterations, and 5 load-balancing-by-dimension iterations. 25

4.7 The mesh for the Level-2 experiment. 27

4.9 The results of the parametric study with no load balancing, 5 original load balancing iterations, and 5 load-balancing-by-dimension iterations for the Level-2 mesh. .. 27

4.8 A demonstration of the original load balancing and load-balancing-by-dimension algorithms on the Level-2 mesh partitioned into 7x7 subsets. 30

5.1 An undirected graph with 4 nodes and 4 edges. 33

5.2 A directed graph with 4 nodes and 4 edges. 33

5.3 A directed graph with a cycle between nodes 1 and 3. 34

5.4 A weighted directed acyclic graph with 4 nodes and 4 weighted edges. 35

5.5 A 3x3 subset partitioning scheme and its corresponding adjacency matrix. 37

5.6 The quadrant layout for 2D problems. 38

5.7 The upper triangular (left) and lower triangular (right) portions of the adjacency matrix in Fig. 5.5. 39

5.8 The quadrant 0 DAG (left) and the quadrant 3 DAG (right). 39

5.9 The flipped subset ordering and corresponding "flipped" adjacency matrix for the partitioning scheme in Fig. 5.5. 40

5.10 The quadrant 1 DAG (left) and the quadrant 2 DAG(right). 41

5.11 The mesh in Fig. 4.1 partitioned into 7 subsets in each dimension with load-balanced-by-dimension cuts. $f = 1.49$ 45

5.12 The cell count per subset for the mesh in Fig. 5.11 from PDT and the time-to-solution estimator. 46

5.13 The graphs for the first (left) and second (right) anglesets 47

5.14 A TDG before (left) and after (right) universal edge weighting is applied. 48

5.15 Examples of regular partitioning. 51

5.16 A 2D verification suite with regular partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 anglesets per quadrant. 52

5.17 Examples of "mildly random" partitioning. 53

5.18 A 2D verification suite with "mildly random" partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 anglesets per quadrant. 54

5.19 Examples of "random" partitioning. 55

5.20 A 2D verification suite with "random" partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 anglesets per quadrant. 56

5.21 Examples of probable worst-case partitioning. 57

5.22 A 2D verification suite with probable worst-case partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 angles per quadrant. 58

5.23 A 3D verification suite with regular partitions run from 2^3 to 10^3 subsets with each case being run from 1 to 6 angles per octant. 59

5.24 The stage counts of PDT's performance model and the time-to-solution estimator for the scaling suite in Table 5.2. 62

5.25 The time per sweep of PDT, the PDT performance model, and the time-to-solution estimator. 62

5.26 The parallel efficiency relative to 8 cores of PDT, the PDT performance model, and the time-to-solution estimator. 63

5.27 The sweep times of the time-to-solution estimator and PDT for the unbalanced pin mesh for 2 to 10 subsets in each dimension with regular cuts. 65

5.28 The sweep times of the time-to-solution estimator and PDT for the mesh in Fig. 4.1 for 2 to 10 subsets in each dimension with load-balanced cuts. 67

5.29 The sweep times of the time-to-solution estimator and PDT for the mesh in Fig. 4.1 for 2 to 10 subsets in each dimension with load-balanced-by-dimension cuts. 68

5.30 A more refined version of the unbalanced pin mesh, containing 9656 cells as opposed to 420 cells. 69

5.31 The sweep times of the time-to-solution estimator and PDT for the more refined unbalanced pin mesh for 2 to 10 subsets in each dimension with regular cuts 70

5.32 The Level-2 experiment mesh evenly partitioned into 42 subsets in x and 13 subsets in y. $f = 32.616$. 71

5.33 The Level-2 experiment mesh partitioned and manually load balanced with 42 subsets in x and 13 subsets in y. $f = 2.386$. 72

6.1 An unstructured mesh with natural boundaries at 1 cm intervals in both dimensions. 75

6.2 The x-vertex CDF of the mesh shown in Fig. 6.1 76

6.3 The derivative of the CDF shown in Fig. 6.2. 77

6.4 A binary tree where each node represents the number of columns we are attempting to find a natural boundary through. 79

6.5 The mesh for the Level-2 experiment (same as Fig. 4.7). 80

6.6 The Level-2 Mesh partitions at three stages of the choosing the optimization cut process. 81

6.7 The time-to-solution and load-balance metric for binary tree, regular, load-balanced and load-balanced-by-dimension cuts on the unbalanced pins mesh from 2 to 10 subsets in each dimension. 82

6.8 The time-to-solution and load-balance metric for binary tree, regular, load-balanced and load-balanced-by-dimension cuts on the "heavier" unbalanced pins mesh from 2 to 10 subsets in each dimension. 83

6.9 The Level-2 mesh sweep times from the time-to-solution estimator for (1) regular cuts, (2) hand-balanced cuts, (3) load-balanced cuts, (4) load-balanced-by-dimension cuts, and (5) optimized cuts. Additionally, the load-balance metric f is reported for each partition type. 84

6.10 The IM1C experiment mesh run through the time-to-solution estimator for regular, load-balanced-by-dimension, and binary tree cuts with 5 subsets in each dimension. Additionally, the load-balance metric f is reported for each partition type. 85

A.1 The time-to-solution and load-balance metric for binary tree, regular, load-balanced and load-balanced-by-dimension cuts on the unbalanced pins mesh from 2 to 10 subsets in each dimension. 93

A.2 The time-to-solution and load-balance metric for binary tree, regular, load-balanced and load-balanced-by-dimension cuts on the "heavier" unbalanced pins mesh from 2 to 10 subsets in each dimension. 94

A.3 The Level-2 mesh sweep times from the time-to-solution estimator for (1) regular cuts, (2) hand-balanced cuts, (3) load-balanced cuts, (4) load-balanced-by-dimension cuts, and (5) optimized cuts. Additionally, the load-balance metric f is reported for each partition type. 94

LIST OF TABLES

TABLE Page

4.1 The tabulated results of the parametric study shown in Fig. 4.6 with no load balancing, 5 original load balancing iterations (LB), and 5 load-balancing-by-dimension (LBD) iterations. 25

4.2 The percent decrease of f with the original load balancing algorithm (LB) and the load-balancing-by-dimension algorithm (LBD) relative to no load balancing, and the percent decrease of f with the load-balancing-by-dimension algorithm relative to the original load balancing algorithm. 26

4.3 The tabulated results of the parametric study shown in Fig. 4.9 with no load balancing, 5 original load balancing iterations (LB), and 5 load-balancing-by-dimension (LBD) iterations for the Level-2 mesh. 28

4.4 The percent decrease of f with the original load balancing algorithm (LB) and the load-balancing-by-dimension algorithm (LBD) relative to no load balancing, and the percent decrease of f with the load-balancing-by-dimension algorithm relative to the original load balancing algorithm for the Level-2 mesh. 29

5.1 The cell count per subset for the time-to-solution estimator and PDT for the mesh in Fig. 4.1 partitioned into 2 subsets in each dimension with regular cuts. 44

5.2 The scaling suite parameters ran with 1 energy group, 80 directions, and $A_m = 10$... 60

5.3 The percent difference between (1) PDT and its performance model, (2) PDT's performance model and the time-to-solution estimator, and (3) PDT and the time-to-solution estimator. 61

5.4 The percent difference in sweep times between the time-to-solution estimator and PDT for the sweep times shown in Fig. 5.27. 66

5.5 The percent difference in sweep times between the time-to-solution estimator and PDT for the sweep times shown in Fig. 5.28. 67

5.6 The percent difference in sweep times between the time-to-solution estimator and PDT for the sweep times shown in Fig. 5.29. 68

5.7 The percent difference in sweep times between the time-to-solution estimator and PDT for the sweep times shown in Fig. 5.31. 71

5.8 The sweep times for the regular cut and manually balanced cut Level-2 problems for PDT and the time-to-solution estimator. 72

5.9 The percent difference for the regular cut and manually balanced cut Level-2 problems between PDT and the time-to-solution estimator. 72

1. INTRODUCTION

The field of radiation transport studies the distribution of radiation (particle/energy) throughout a seven-dimensional phase-space consisting of time, space, energy, and direction. Radiation transport is commonly used in reactor physics, medical physics, radiation shielding, criticality safety applications, stellar atmospheres, as well as the general simulation of radiation in different environments.

Radiation transport is described by the Boltzmann transport equation [1, 2, 3, 4] and can be solved stochastically or deterministically. The Monte Carlo method is used for stochastic radiation transport by tracking the behavior of a finite number of particles from their "birth" (for example, a neutron source) until their "death" (getting absorbed or escaping through a problem boundary), and statistically evaluating relevant quantities of interest (for example, neutron flux in a detector) [5]. Codes such as MCNP [6] and SHIFT [7] utilize the Monte Carlo method to perform high-fidelity radiation transport simulations for a variety of experiments and problems.

Deterministic transport discretizes the time, space, energy, and angle variables, yielding an extremely large system of linear equations. This system is then solved iteratively to obtain a numerical solution to the transport equation. There are various discretization techniques to yield the linear system of equations; e.g., the method of characteristics, collision probabilities, discrete-ordinates, spectral methods etc. Some of them are implemented by codes such as MPACT [8], PARTISN [9], Denovo [10], and PDT [11, 12, 13].

The work presented in this dissertation deals with the transport equation discretized using multigroup in energy, discrete-ordinates (collocation) in angle, and discontinuous finite elements in space. The resulting discrete system of equations is amenable to iterative techniques whereby the streaming+total interaction operator need not to be assembled, generating large memory savings while being computationally very efficient. Essentially, this allows for a matrix-free approach, termed "transport sweeps" in the radiation transport community, where small local system of equations are solved cell-by-cell, for each angular direction and energy group, in a fashion that follows

the flow of radiation in the computational mesh. Because the solution process marches through the mesh, it is known as sweeping.

It is often desirable to split the problem domain so that the memory cost can be distributed across multiple processors. Extensive research has been done in parallelizing the transport sweep [14, 9, 10, 11, 12, 13], as the transport sweep gains complexity for distributed meshes. Maintaining sweep performance for massively parallel simulations has and continues to be a thriving research topic.

Research on sweeping on unstructured meshes has come to prominence in recent years, as the need to simulate more complex problems has risen. Partitioning unstructured meshes in a balanced (or near equivalent amount of work per processor) fashion that preserves mesh integrity can present challenges. The following dissertation chapters will discuss a brief history of the transport sweep, parallel transport sweeps, load balancing unstructured meshes for transport, a time-to-solution estimator, and optimizing partitions for parallel transport.

Chapter 2, The Deterministic Transport Sweep, introduces the steady-state neutron transport equation and the discretization process to obtain the discrete form of the transport equation. Chapter 2 also provides more details on the transport sweep.

Massively parallel transport sweeps have been shown to scale up to 1.5 million cores on logically Cartesian grids using Texas A&M's flagship deterministic transport code, PDT [11, 12, 13]. Chapter 3, Parallel Transport Sweeps, describes the basics of a parallel sweep algorithm, and specifically describes the KBA algorithm [14] and PDT's extension of the KBA algorithm. This chapter also describes PDT's performance model, which predicts the sweep time given a set of partitioning parameters.

In order to solve a wider set of problems, an unstructured meshing capability was implemented in PDT. However, unstructured meshes may lead to imbalanced partitions, or different numbers of mesh cells per processor subdomain. Chapter 4, Unstructured Meshing and Load Balancing in PDT, details the two load-balancing algorithms implemented in PDT, the original load-balancing algorithm and the load-balancing-by-dimension algorithm [15, 16].

PDT's existing performance model does not account for unstructured meshes and imbalanced partitions. Chapter 5, Time-to-Solution Estimator, describes the time-to-solution estimator, a graph-based method that estimates the time it takes to sweep across a problem given a mesh, partitioning scheme, and machine-specific parameters.

The time-to-solution estimator described in Chapter 5 is used to optimize the partitioning scheme. Chapter 6, Partitioning Optimization, details two optimization methods: a "black box" method that uses Python's scipy.optimize library and a "human-intelligence" method that prioritizes partition placement in locations that do not add cells to a mesh.

2. THE TRANSPORT SWEEP

The steady-state neutron transport equation describes the behavior of neutrons in a medium and is given by Eq. (2.1):

$$\vec{\Omega} \cdot \vec{\nabla}\psi(\vec{r}, E, \vec{\Omega}) + \Sigma_t(\vec{r}, E)\psi(\vec{r}, E, \vec{\Omega}) = \int_0^\infty dE' \int_{4\pi} d\Omega' \Sigma_s(\vec{r}, E' \to E, \Omega' \to \Omega)\psi(\vec{r}, E', \vec{\Omega}') + S_{ext}(\vec{r}, E, \vec{\Omega}), \quad (2.1)$$

where $\vec{\Omega} \cdot \vec{\nabla}\psi$ is the leakage term and $\Sigma_t\psi$ is the total collision term (absorption, outscatter, and within group scattering). These represent the loss terms of the neutron transport equation. The right hand side of Eq. (2.1) represents the gain terms, where S_{ext} is the external source of neutrons and $\int_0^\infty dE' \int_{4\pi} d\Omega' \Sigma_s(E' \to E, \Omega' \to \Omega)\psi(\vec{r}, E', \vec{\Omega}')$ is the inscatter term, which represents all neutrons scattering from energy E' and direction $\vec{\Omega}'$ into dE about energy E and $d\Omega$ about direction $\vec{\Omega}$.

Without loss of generality for the problem at hand, we assume isotropic scattering for simplicity. The double differential scattering cross section, $\Sigma_s(E' \to E, \Omega' \to \Omega)$, has its angular dependence present in the integral, and is divided by 4π to reflect isotropic behavior. This yields:

$$\begin{aligned} &\vec{\Omega} \cdot \vec{\nabla}\psi(\vec{r}, E, \vec{\Omega}) + \Sigma_t(\vec{r}, E)\psi(\vec{r}, E, \vec{\Omega}) \\ &= \frac{1}{4\pi} \int_0^\infty dE' \Sigma_s(\vec{r}, E' \to E) \int_{4\pi} d\Omega' \psi(\vec{r}, E', \vec{\Omega}') + S_{ext}(\vec{r}, E, \vec{\Omega}) \\ &= \frac{1}{4\pi} \int_0^\infty dE' \Sigma_s(\vec{r}, E' \to E)\phi(\vec{r}, E') + S_{ext}(\vec{r}, E, \vec{\Omega}), \end{aligned} \quad (2.2)$$

where we have introduced the scalar flux as the integral of the angular flux:

$$\phi(\vec{r}, E') = \int_{4\pi} d\Omega' \psi(\vec{r}, E', \vec{\Omega}'). \quad (2.3)$$

The next step to solving the transport equation is to discretize in energy, yielding Eq. (2.4), the

multigroup transport equation:

$$\vec{\Omega}\cdot\vec{\nabla}\psi_g(\vec{r},\vec{\Omega})+\Sigma_{t,g}(\vec{r})\psi_g(\vec{r},\vec{\Omega}) = \frac{1}{4\pi}\sum_{g'}\Sigma_{s,g'\to g}(\vec{r})\phi_{g'}(\vec{r})+S_{ext,g}(\vec{r},\vec{\Omega}), \quad \text{for } 1\leq g\leq G \tag{2.4}$$

where the multigroup transport equations now form a system of coupled equations.

Next, we discretize in angle using the discrete ordinates method [10], whereby an angular quadrature $\left(\vec{\Omega}_m, w_m\right)_{1\leq m\leq M}$ is used to solve the above equations along a given set of directions $\vec{\Omega}_m$:

$$\vec{\Omega}_m\cdot\vec{\nabla}\psi_{g,m}(\vec{r})+\Sigma_{t,g}(\vec{r})\psi_{g,m}(\vec{r}) = \frac{1}{4\pi}\sum_{g'}\Sigma_{s,g'\to g}(\vec{r})\phi_{g'}(\vec{r})+S_{ext,g,m}(\vec{r}), \tag{2.5}$$

where the subscript m is introduced to denote the angular flux in direction m. In order to evaluate the scalar flux, we employ the angular weights w_m and the angular flux solutions ψ_m to numerically perform the angular integration:

$$\phi_g(\vec{r})\approx\sum_{m=1}^{m=M} w_m\psi_{g,m}(\vec{r}). \tag{2.6}$$

At this point, it is clear that we are solving a sequence of transport equations for one group and direction at a time. Therefore, all transport equations take the following form:

$$\vec{\Omega}_m\cdot\vec{\nabla}\psi_m(\vec{r})+\Sigma_t(\vec{r})\psi_m(\vec{r}) = \frac{1}{4\pi}\Sigma_s(\vec{r})\phi(\vec{r})+q_m^{ext+inscat}(\vec{r}) = q_m(\vec{r}), \tag{2.7}$$

where the group index is omitted for brevity.

In order to obtain the solution for this discrete form of the transport equation, an iterative process, source iteration, is introduced. This is shown in Eq. (2.8):

$$\vec{\Omega}_m\cdot\vec{\nabla}\psi_m^{(l+1)}(\vec{r})+\Sigma_t\psi_m^{(l+1)}(\vec{r}) = q_m^{(l)}(\vec{r}), \tag{2.8}$$

where the right hand side terms of Eq. (2.5) have been combined into one general source term, q_m. The angular flux of iteration $(l+1)$ is calculated using the (l^{th}) value of the total source.

After the angular and energy dependencies have been accounted for, Eq. (2.8) must be discretized in space as well. This is done by meshing the domain and utilizing one of three popular discretization techniques: finite difference [17], finite volume [17], or discontinuous finite element [18], allowing one cell at a time to be solved. The solution across a cell interface is connected based on an upwind approach, where face outflow radiation becomes face inflow radiation for the downwind cells. Figure 2.1 shows the sweep ordering for a given direction on both a structured and unstructured mesh.

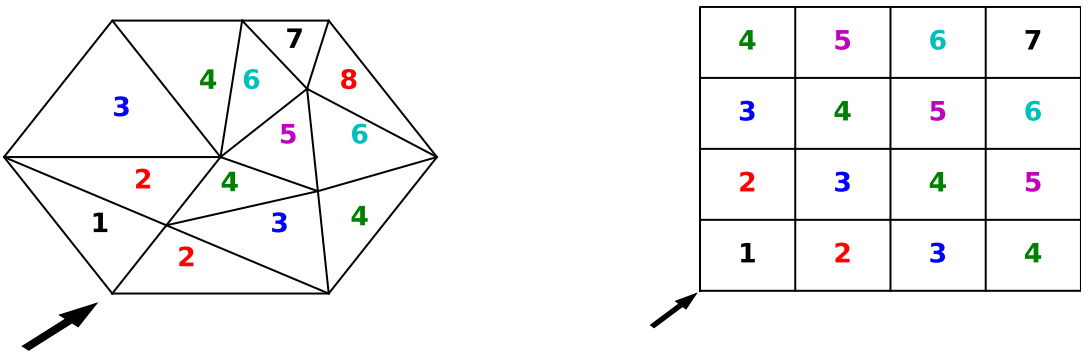


Figure 2.1: A demonstration of a sweep on structured and unstructured meshes.

The number in each cell represents the order in which the cells are solved. All cells must receive the solution from downwind cells before their solution can be obtained.

The transport sweep can be parallelized in order to mitigate memory costs and obtain solutions in a reasonable time for large problems. Chapter 3 describes the basics of a parallel sweep algorithm and details two algorithms: The KBA algorithm [14] and PDT's extension of the KBA algorithm [11, 12, 13].

3. PARALLEL TRANSPORT SWEEPS

As mentioned in Chapter 2, a transport sweep is set up by overlaying a domain with a finite element mesh and solving the transport equation cell by cell using a discontinuous finite element approach. A transport sweep can be solved in parallel in order to obtain the solution faster, as well as distribute the problem across many processors for memory intensive problems.

In PDT, a transport sweep can be performed on a structured Cartesian or arbitrary polyhedral mesh. Sweeping on an unstructured mesh presents two challenges: maintaining sweep efficiency on a massively parallel scale and keeping non-concave sub-domains to avoid cycles. PDT has already proven the ability to perform massively parallel transport sweeps on structured meshes. As part of previous efforts in PDT, researchers have outlined three important properties for parallel sweeps.

A parallel sweep algorithm is defined by three properties [11] :

- partitioning: dividing the domain among available processors,
- aggregation: grouping cells, directions, and energy groups into tasks,
- scheduling: choosing which task to execute if more than one is available.

It is important to note that these properties apply after any cell-level cycles are broken, but partitioning can introduce cycles between processors.

The basic concepts of parallel transport sweeps, partitioning, aggregation, and scheduling, are most easily described in the context of a transport sweep that takes place on a structured Cartesian mesh. Furthermore, the work detailed in Chapters 4 and 5 utilize aspects of the structured transport sweep.

In a regular grid, we have the number of cells in each Cartesian direction: N_x, N_y, N_z. These cells are aggregated into "cellsets", using aggregation factors A_x, A_y, A_z. If M is the number of angular directions per octant, G is the total number of energy groups, and N is the total number

of cells, then the total fine-grain work units is $8MGN$. The factor of 8 is present as M directions are swept for all 8 octants. We often discuss the directions in a sweep in terms of octant-pairs, or two octants that have opposing sweep ordering. This is an important concept that is discussed in Section 3.1. The finest-grain work unit is the calculation of a single direction and energy group's unknowns in a single cell, or $\psi_{m,g}$ for a single cell.

Fine-grain work units are aggregated into coarser-grained units called *tasks*. A few terms are defined that describe how each variable is aggregated.

- $A_x = \frac{N_x}{P_x}$, where N_x is the number of cells in x and P_x is the number of processors in x.
- $A_y = \frac{N_y}{P_y}$, where N_y is the number of cells in y and P_y is the number of processors in y.
- A_z = a selectable number of z-planes of $A_x A_y$ cells.
- $N_g = \frac{G}{A_g}$
- $N_m = \frac{M}{A_m}$
- $N_k = \frac{N_z}{P_z A_z}$
- $N_k A_x A_y A_z = \frac{N_x N_y N_z}{P_x P_y P_z}$

It follows that each process owns N_k cellsets (each of which is A_z planes of $A_x A_y$ cells), $8N_m$ angle-sets, and N_g group-sets for a total of $8N_m N_g N_k$ tasks.

One task contains $A_x A_y A_z$ cells, A_m directions, and A_g groups. Equivalently, a task is the computation of one cellset, one groupset, and one angleset, with the completion of one task defined as a stage. The stage is particularly important when assessing sweeps against analytical performance models. Equation (3.1) defines parallel sweep efficiency:

$$\begin{aligned} \epsilon &= \frac{T_{\text{task}} N_{\text{tasks}}}{[N_{\text{stages}}][T_{\text{task}} + T_{\text{comm}}]} \\ &= \frac{1}{[1 + \frac{N_{\text{idle}}}{N_{\text{tasks}}}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]}, \end{aligned} \tag{3.1}$$

where N_{idle} is the number of idle stages per processor, N_{tasks} is the number of tasks each processor performs, T_{comm} is the time to communicate after completion of a task, and T_{task} is the time it takes to compute one task. Equations (3.3) and (3.4) show how T_{comm} and T_{task} are calculated:

$$T_{\text{comm}} = M_L T_{\text{latency}} + T_{\text{byte}} N_{\text{bytes}}, \tag{3.2}$$

$$N_{\text{bytes}} = (A_x A_y + A_x A_z + A_y A_z) A_g A_m upbc, \tag{3.3}$$

$$T_{\text{task}} = A_x A_y A_z A_m A_g T_{\text{grind}}, \tag{3.4}$$

where T_{latency} is the message latency time, T_{byte} is the time required to send one byte of message, N_{bytes} is the total number of bytes of information that a processor must communicate to its downstream neighbors at each stage, $upbc$ is the unknowns per boundary cell (generally 2 for 2D and 4 for 3D) and T_{grind} is the time it takes to compute a single cell, direction, and energy group. M_L is a latency parameter that is used to explore performance as a function of increased or decreased latency. Note that Eq. 3.4 is idealized as it does not take into account overhead in various parts of the sweep implementation.

Before expanding on the proposed method of partitioning and scheduling for parallel transport sweeps, a quick review of two parallel transport sweep algorithms is necessary. This dissertation's method will expand on PDT's sweep algorithm [11, 12, 13], which is an extension of the popular KBA algorithm [14].

3.1 The KBA Algorithm

Several parallel transport sweep codes use KBA partitioning in their sweeping, such as Denovo [10] and PARTISN [9]. The KBA partitioning scheme and algorithm was developed by Koch, Baker, and Alcouffe [14].

The KBA algorithm traditionally chooses $P_z = 1, A_m = 1$ (or 1 angle per angleset), $G = A_g = 1, A_x = N_x/P_x, A_y = N_y/P_y$, with A_z being the selectable number of z-planes to be aggregated into each task. This partitions the domain into long, thin columns. With $N_k = N_z/A_z$,

each processor performs $N_{\text{tasks}} = 8MN_k$ tasks. The KBA algorithm traditionally solves a problem one energy group at a time, and uses a “pipelining” or assembly line approach. This allows for new angles in an octant to begin sweeping before old angles have fully swept across the spatial domain. Figure 3.1 shows an example of pipelining the angular work of a quadrant. Figure 3.2 shows pipelining in the X-Z plane with $A_z = 1$.

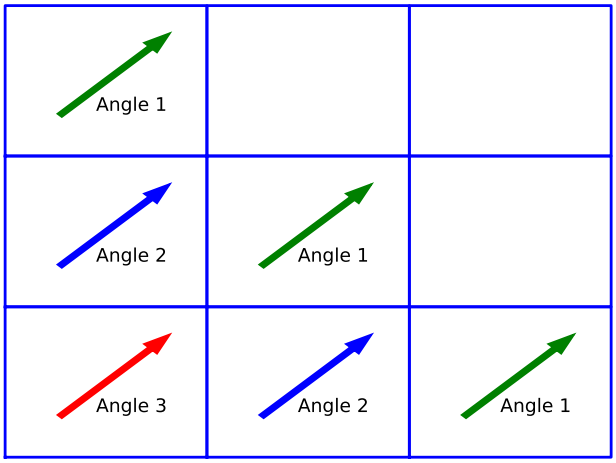


Figure 3.1: An example showing the pipelining of angular work from the lower left quadrant.

In Fig. 3.1, each square represents a processor that owns a group of A_xA_y cells. We see that as soon as a processor is free to solve the next angle with the same sweep ordering, it begins immediately. Figure 3.2 shows us the X-Z plane of a processor layout with $P_x = 2$ and $P_z = 1$. We see $N_k = \frac{N_z=5}{A_z=1} = 5$ planes in Z, and three angles pipelined. We notice as soon as a cellset of $A_xA_yA_z$ cells is free to solve the next angle with the same sweep ordering, it begins immediately. KBA introduced pipelining in order to combat the inherent inefficiencies of waiting for all processors to complete a sweep in a direction before starting the next angle.

There are two variants to the KBA algorithm, “successive in angle, successive in quadrant", and “simultaneous in angle, successive in quadrant". With “successive in angle, successive in

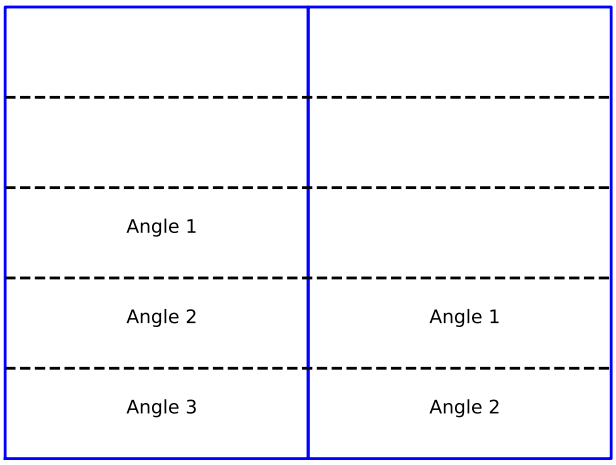


Figure 3.2: An example showing pipelining in the X-Z plane with $A_z = 1$, $P_x = 2$ and $P_z = 1$.

quadrant", an octant pipelines its angular work, and once all directions are complete the opposing octant pipelines them back. This is then done for the remaining octant pairs. With “simultaneous in angle, successive in quadrant", all angles from one octant are aggregated and solved rather than pipelined, and upon completion the opposing octant solves them. This is done either by aggregating angles, This is then done for the remaining octant pairs. The KBA parallel efficiency [11] for “successive in angle, successive in quadrant" is:

$$\epsilon_{KBA} = \frac{1}{[1 + \frac{4(P_x+P_y-2)}{8MN_k}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]}. \tag{3.5}$$

The KBA parallel efficiency for “simultaneous in angle, successive in quadrant” is:

$$\epsilon_{KBA} = \frac{1}{[1 + \frac{4(P_x+P_y-2)}{8MN_k/(A_m)}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]}. \tag{3.6}$$

For each octant-pair, the idle time is $P_x + P_y - 2$, and if the successive octant-pairs don’t start until the previous octant pair finishes, then the total idle time is $4(P_x + P_y - 2)$.

3.2 PDT's Extension of KBA

PDT's extension of KBA does not limit P_z, A_m, G, or A_g. In addition, all 8 octants (4 quadrants in 2D) begin work immediately. Unlike KBA, PDT's scheduling requires conflict resolution in its algorithm, as the pipelines from all octants will end up colliding toward the middle of the processor domain.

If two or more tasks reach a processor at the same time, PDT employs a tie breaking strategy:

1. The task with the greater depth-of-graph remaining (simply, more work remaining) goes first.

2. If the depth-of-graph remaining is tied, then octant-priority tie-breaking is used:

   (a) The task with $\Omega_x > 0$ wins.

   (b) If multiple tasks have $\Omega_x > 0$, then the task with $\Omega_y > 0$ wins.

   (c) If multiple tasks have $\Omega_y > 0$, then the task with $\Omega_z > 0$ wins.

Given these conflict resolution techniques, the minimum possible number of stages for given partitioning parameters P_i and A_j is $2N_{\text{fill}} + N_{\text{tasks}}$. N_{fill} is both the minimum number of stages before a sweepfront can reach the center-most processors and the number needed to finish a direction's sweep after the center-most processors have finished. Equations 3.7, 3.8, and 3.9 define N_{fill}, N_{idle}, and N_{tasks}:

$$N_{\text{fill}} = \frac{P_x + \delta_x}{2} - 1 + \frac{P_y + \delta_y}{2} - 1 + N_k(\frac{P_z + \delta_z}{2} - 1) \tag{3.7}$$

$$\delta_u = 0 \text{ or } 1 \text{ for } P_u \text{ even or odd, respectively}$$

$$N_{\text{idle}} = 2N_{\text{fill}} \tag{3.8}$$

$$N_{\text{tasks}} = 8N_m N_g N_k \tag{3.9}$$

Plugging these definitions into Eq. 3.1, the PDT optimal parallel efficiency [11] for P_u even is:

$$\epsilon_{opt} = \frac{1}{[1 + \frac{P_x+P_y-4+N_k(P_z-2)}{8MGN_k/(A_m A_G)}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]} \tag{3.10}$$

Let us consider how many more processors the hybrid partitioning with optimal scheduling ($P_z = 2$) can use while yielding the same efficiency as KBA. Consider the limit of a large number of processors for both KBA and hybrid partioning with an optimal schedule. Equation 3.11 shows the KBA efficiency in the large-P limit with $P_x + P_y \approx 2P^{1/2}$. Setting $A_m = A_g = 1$ to match traditional KBA aggregation parameters, Eq. 3.12 shows the hybrid optimal efficiency in the large-P limit with $P_z = 2$ and $P_x + P_y \approx 2P^{1/2}$.

$$\epsilon_{KBA} \xrightarrow{\text{large P}} \frac{1}{[1 + \frac{(4P^{1/2})}{8MN_k}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]} \tag{3.11}$$

$$\epsilon_{opt,hybrid} \xrightarrow{\text{large P}} \frac{1}{[1 + \frac{\sqrt{2}P^{1/2}}{8MN_k}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]} \tag{3.12}$$

If we set Eq. 3.11 equal to Eq. 3.12, we see that in the limit of a large number of processors P, the optimal algorithm yields the same efficiency as KBA with 32 times as many processors [11, 12, 13].

3.3 PDT's Performance Model

To aid in the selection of optimal processor and aggregation parameters, PDT uses a performance model to estimate the sweep time. This performance model is the basis for the cost function described in Chapter 5. Equation 3.13 calculates the sweep time by multiplying the time it takes to solve each stage by the number of stages. This is then multiplied by a multi-core fudge factor that estimates the performance drop-off from 1 to 8 cores due to parallel overhead.

$$\text{Sweep Time} = \text{mcff}\cdot[\text{num stages}\cdot(T_{wu}+3\cdot\text{latency}\cdot M_L+T_{\text{byte}}\cdot N_{\text{bytes}}+N_{\text{cells}}\cdot(T_c+A_m\cdot(T_m+T_g)))], \tag{3.13}$$

where:

- mcff = the Multi-Core Fudge Factor, a corrective factor that accounts for performance drop-off from 1 to 8 cores,
- num stages = tasks per processor + $2N_{\text{idle}}$,
- tasks per processor = $\frac{N_x}{P_x A_x}\frac{N_y}{P_y A_y}\frac{N_z}{P_z A_z}\frac{N_m}{A_m}\frac{N_g}{A_g}$,
- T_{wu} = the time to get into the sweep operator,
- latency = the machine specific communication latency,
- M_L = the machine specific latency multiplier,
- T_{byte} = the communication time per double,
- N_{byte} = the bytes per 3 communications (assuming 3 neighbors) = $(A_x A_y + A_x A_z + A_y A_z) A_g A_m upbc$,
- $upbc$ = unknowns per boundary cell (4 for 3d, 2 for 2d),
- N_{cells} = the number of cells per task, $A_x A_y A_z$,
- T_c = the time spent solving cell-specific work,
- T_m = the time spent solving angle-specific work,
- T_g = the time spent solving group-specific work.

A comparison between this performance model and the time-to-solution estimator will be shown in Chapter 5. Before this, we motivate the need for the time-to-solution estimator in Chapter 4 by describing unstructured meshes in PDT and how we load balance them.

4. UNSTRUCTURED MESHING AND LOAD BALANCING IN PDT

Initially, PDT only swept on structured, logically Cartesian meshes. As the need to solve problems with more complex geometries arose, PDT added a support for arbitrary polyhedral unstructured meshes. However, this introduced imbalanced partitions (or different amounts of cells per processor), causing longer and unmanageable runtimes.

To combat imbalanced partitions, two load balancing algorithms were implemented, referred to in this thesis as the original load-balancing algorithm [15, 16] and the load-balancing-by-dimension algorithm.

Before detailing the two load balancing algorithms PDT employs, a quick review of partitioning unstructured meshes in PDT is necessary:

- "Cut lines" in 2D ("cut planes" in 3D) are used to slice through the mesh in the x, y, and z dimensions.
- The cut planes form brick partitions, called subsets, that have unstructured meshes inside of them.
- Discontinuities along subset boundaries are fixed by "stitching" hanging nodes, creating degenerate polygons along subset boundaries.
- The subsets are distributed amongst the processor domain.

Using cut lines/cut planes to partition unstructured meshes may preserve the provably optimal sweep partitioning [11, 12, 13] of logically Cartesian grids if each partition has an equivalent number of cells. Subsets, rather than the cells, have (i, j, k) indices and will become the base unit when aggregating spatial parameters. That is, A_x and A_y will now represent the number of subsets in x and y aggregated into a task. Generally, one subset is assigned per processor, as this minimizes the communication costs across processor boundaries, although this is not required. Figure 4.1 shows an example of an unstructured mesh partitioned into 100 subsets in PDT.

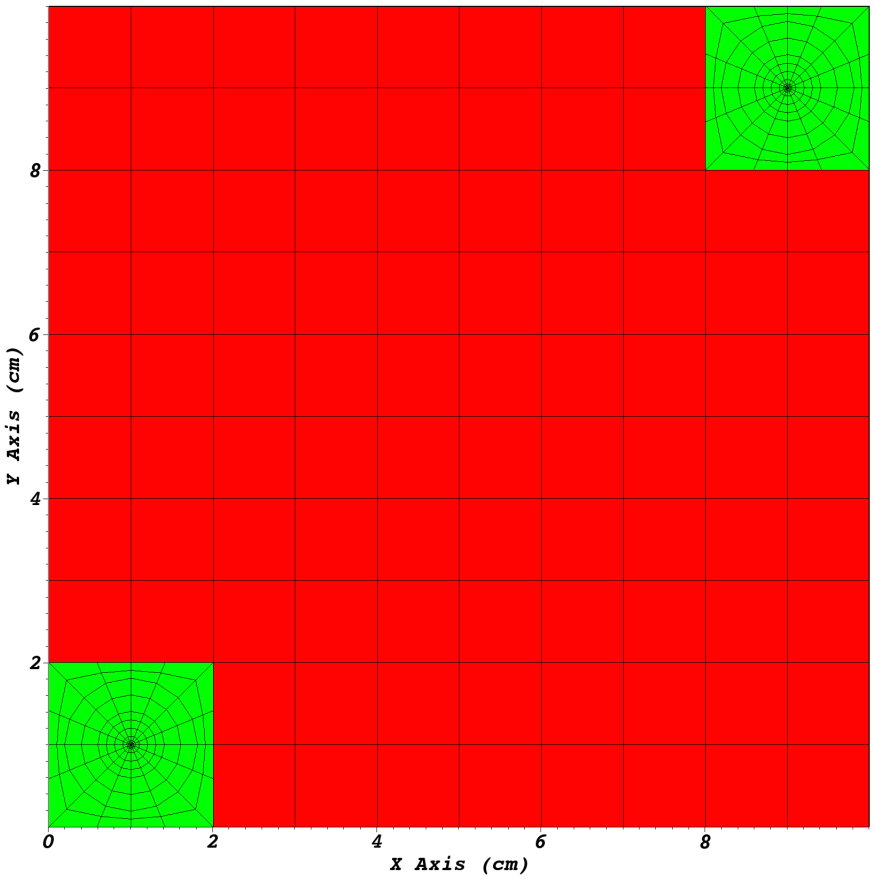


Figure 4.1: An unstructured mesh partitioned into 100 subsets with cut lines at 1 cm intervals in both dimensions

Upon creation, the subsets may have geometric discontinuities as a result of slicing through the mesh. Figure 4.2 shows an example of a hanging node across a subset boundary. This node is stitched across the boundary to preserve geometric continuity, forming a degenerate polygon [19] (in Fig. 4.2, a degenerate square). PDT uses Piece-Wise Linear Discontinuous (PWLD) finite-element basis functions [20, 21] that allow for solutions on arbitrary and degenerate polyhedra.

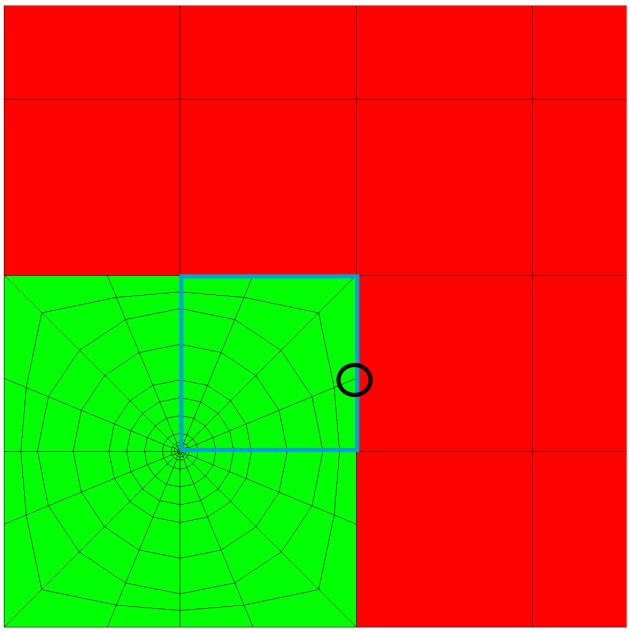


Figure 4.2: A hanging node (circled in black) on a subset boundary (highlighted in blue).

Both approaches to load balancing move cut lines in order to redistribute cells more evenly throughout subsets. We define a metric describing how imbalanced our problem is:

$$f = \frac{\max\limits_{ijk}(N_{ijk})}{\frac{N_{tot}}{I \cdot J \cdot K}}, \tag{4.1}$$

where f is the load balance metric, N_{ijk} is the number of cells in subset (i, j, k), N_{tot} is the global number of cells in the problem, and I, J, and K are the total number of subsets in the x, y, and z directions, respectively. The metric is a measure of the maximum number of cells per subset divided by the average number of cells per subset. For a perfectly balanced problem, $f = 1$.

Figure 4.3 illustrates an example of redistributing the cut planes in x to balance the cells per column.

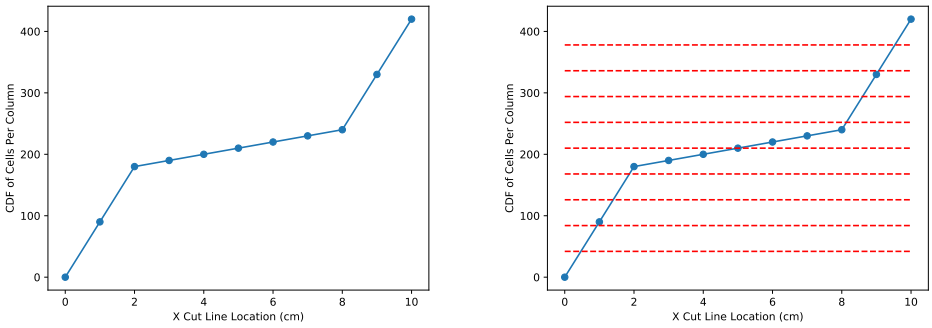


Figure 4.3: The use of the CDF of cells per column to redistribute the cut lines in X.

The image on the left side of Fig. 4.3 shows the CDF of the cells per column in Fig. 4.1. The red lines on the right side of Fig. 4.3 show the ideal equal number of cells per column. The x-value of the intersection of these red lines and the CDF are where the cut lines are redistributed to.

In order to decide the necessity of redistributing a dimension's cut lines/planes, we use dimensional sub-metrics of the following form:

$$f_Z = \frac{\max\limits_k [\sum_{i,j} N_{ijk}]}{\frac{N_{tot}}{K}}, \tag{4.2}$$

where K is the total number of z-planes. Equation 4.2 is a metric defining how imbalanced the problem's planes are. It calculates the maximum cells per plane divided by the average cells per plane. If f_K is greater than a predefined tolerance, the z cut planes are redistributed using the process in Fig. 4.3.

4.1 Original Load-Balancing Algorithm

The initial approach to load balancing was implemented on 2D extruded meshes, meaning the mesh is balanced in the 2D plane and then extruded, yielding a balanced 3D mesh. The metrics for

this algorithm are defined as follows:

$$f = \frac{\max_{ij}(N_{ij})}{\frac{N_{tot}}{I \cdot J}} \tag{4.3}$$

$$f_X = \frac{\max_{i}[\sum_j N_{ij}]}{\frac{N_{tot}}{I}} \tag{4.4}$$

$$f_Y = \frac{\max_{j}[\sum_i N_{ij}]}{\frac{N_{tot}}{J}} \tag{4.5}$$

Equation 4.3 mirrors Eq. 4.1 for 2 dimensions, and Eqs. 4.4 and 4.5 define the column and row-wise metrics respectively.

Algorithm 1 summarizes the original approach to load balancing meshes in PDT.

Algorithm 1 The original load-balancing algorithm.

while $f > 1 + \text{tol}_{\text{subset}}$ **do**
  if $f_X > 1 + \text{tol}_{\text{col}}$ **then**
    Redistribute the X cut lines.
  end if
  if $f_Y > 1 + \text{tol}_{\text{row}}$ **then**
    Redistribute the Y cut lines.
  end if
end while

While the problem is not balanced:

- Check if the columns are balanced, and if not redistribute the X cut lines.
- Check if the rows are balanced, and if not redistribute the Y cut lines.
- Repeat until the mesh is balanced or until a maximum number of iterations is reached.

The original load-balancing algorithm placed cut lines in all dimensions all the way through the mesh. This created an orthogonal partitioning where each subset had an equivalent number of neighbors, which was done to preserve the provably optimal sweep partitioning described by Adams et. al [11, 12, 13]. However, there are theoretical limits to load balancing in this fashion. Figure 4.4 shows a simple 2D subset layout with M unaligned patches with N cells each.

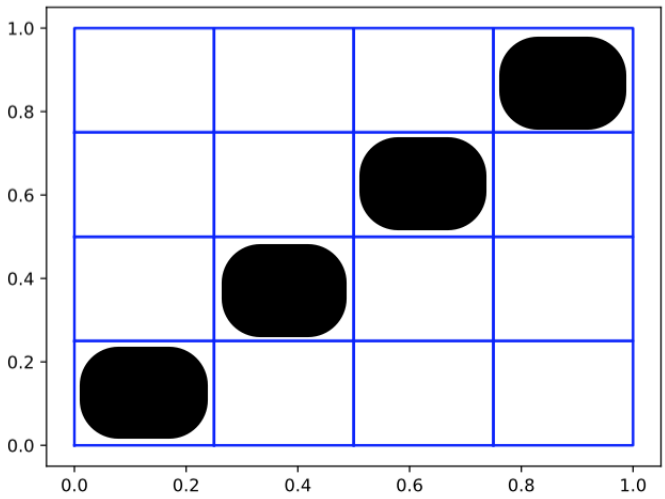


Figure 4.4: A 2D subset layout with M unaligned patches of high mesh density N.

The subset layout is M^2, but only M subset have significant work, leading to a theoretical limit for the load imbalance factor:

$$f = \frac{N}{(MN + C)/M^2} \xrightarrow{N \to \infty} \frac{N}{N/M} = M. \tag{4.6}$$

Due to this theoretical limit, the load-balancing-by-dimension algorithm was developed.

4.2 Load-Balancing-by-Dimension Algorithm

The load-balancing-by-dimension by dimension (LBD) algorithm, similar to the original load-balancing algorithm, relies on the movement of cut lines/planes to redistribute mesh cells in a more balanced manner. However, cut lines are no longer required to go all the way through the

mesh, and the load-balancing-by-dimension algorithm is fully extensible to 3 dimensions. The load-balancing-by-dimension algorithm is summarized by:

1. Slice the mesh in z and redistribute cut planes until each plane has approximately an equivalent number of cells.

2. For each z layer, slice the layer in columns and redistribute the x cut lines until each column has an approximately equivalent number of cells.

3. For each column within each z layer, slice the column in rows and redistribute the y cut lines until each row has an approximately equivalent number of cells.

We once again use dimensional sub-metrics to determine whether or not a dimension's cut lines/planes need to be redistributed. The z dimension's sub-metric is defined by Eq. 4.2. For the LBD algorithm, there are K column-wise metrics, one for each z layer:

$$f_{X,k} = \frac{\max_i [\sum_j N_{ijk}]}{\frac{N_{tot,k}}{I}}, \tag{4.7}$$

where $N_{tot,k}$ is the number of cells in layer k. Equation 4.7 defines the column-wise metric for layer k, or the maximum number of cells per column in layer k divided by the average number of cells per column in layer k.

For the LBD algorithm, there are $K \cdot I$ row-wise metrics, one for each column in each z layer:

$$f_{Y,k,i} = \frac{\max_j N_{ijk}}{\frac{N_{tot,k,i}}{J}}, \tag{4.8}$$

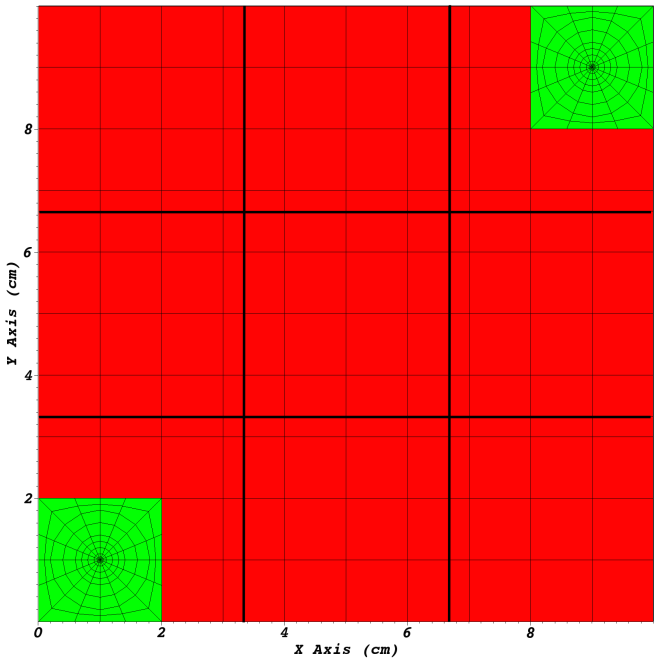
where $N_{tot,k,i}$ is the number of cells in column i in layer k. Equation 4.8 defines the row-wise metric for layer k in column i, or the maximum number of cells per row in column i in layer k divided by the average number of cells per row in column i in layer k.

Algorithm 2 details the load-balancing-by-dimension algorithm.

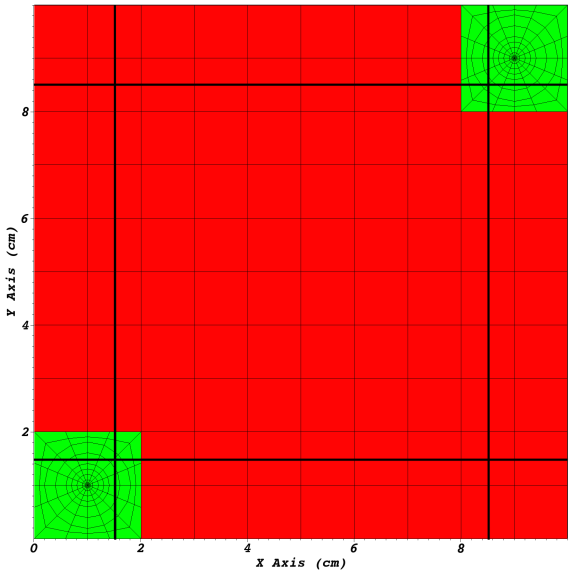
Algorithm 2 The load-balancing-by-dimension algorithm.

```
while f_Z > 1 + tol_K do
    Redistribute the Z cut planes.
end while
for k in K do
    while f_{X,k} > 1 + tol_I do
        Redistribute the X cut lines within layer k.
    end while
end for
for k in K do
    for i in I do
        while f_{Y,k,i} > 1 + tol_J do
            Redistribute the Y cut lines in column i in layer k.
        end while
    end for
end for
Calculate f.
```

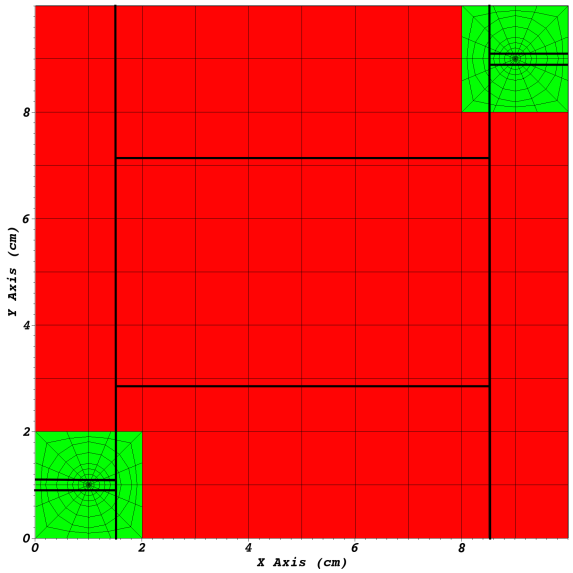
Figure 4.5 illustrates the behavior of both algorithms after 5 iterations. In Fig. 4.5b, we see the partitions cutting across the entire domain, with the x and y cut lines moving into the denser geometric features in the corners to more evenly distribute cells. In Fig. 4.5c, we see the x partitions cutting across the entire domain, but the y partitions being redistributed by column. The y partitions are moved into the respective geometric features in the appropriate columns in order to better balance the problem.



(a) No load balancing, $f = 3.41$.



(b) 5 load balancing iterations, $f = 2.58$.



(c) 5 load-balancing-by-dimension iterations, $f = 1.49$.

Figure 4.5: A demonstration of the original load balancing and load-balancing-by-dimension algorithms on an unstructured mesh partitioned into 3x3 subsets.

4.3 Parametric Study of the Original Load Balancing and the Load-Balancing-by-Dimension Algorithms

To study the behavior and effectiveness of both the original load balancing and the load-balancing-by-dimension algorithm, a parametric study was run on the mesh shown in Fig. 4.1. This mesh was chosen because it is inherently imbalanced as there are two dense geometric features in opposing corners with a sparse region in between.

4.3.1 Parametric study on the unbalanced pin mesh

The study calculates f for the mesh with no load balancing iterations, 5 original load balancing iterations, and 5 load-balancing-by-dimension iterations. The mesh is partitioned into 2 to 10 subsets in each dimension for all load balancing formats. Figure 4.6 shows the results of this parametric study. The results are tabulated in Table 4.1.

With regular cut lines going through the mesh, f_{reg} generally increases as the number of subsets increases. As the number of cut lines increases, the number of cells added may also increase, driving up the maximum number of cells in each subset. It is important to note that in Eq. 4.1, the average number of cells per subset used to calculate f is the average number of cells per subset *before* slicing through the mesh. With more cut lines, the maximum number of cells per subset will increase while the average number of cells per subset will decrease, leading to an increase in f. There can be exceptions to this trend, as seen by f_{reg} for 8 and 10 subsets in each dimension. The cut lines with 10 subsets in each dimension lie on natural boundaries when evenly distributed. A natural boundary is a mesh boundary that coincides with a subset boundary, adding no cells to the mesh. Because no cells are added in the 10x10 case, f_{reg} with 10 subsets in each dimension is lower than f_{reg} with 8 subsets in each dimension.

For the load balanced and load-balanced-by-dimension cases, we see a similar increasing trend for f, although it is not quite as consistent for the load-balanced-by-dimension cases.

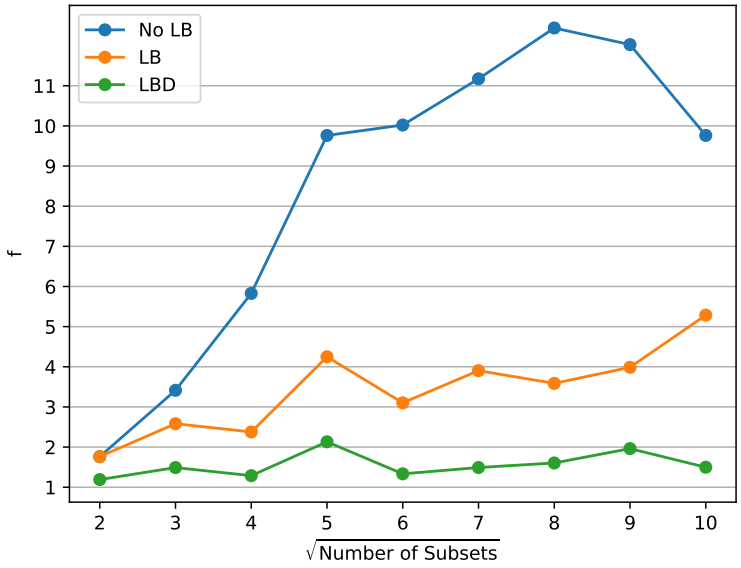


Figure 4.6: The results of the parametric study with no load balancing, 5 original load balancing iterations, and 5 load-balancing-by-dimension iterations.

Table 4.1: The tabulated results of the parametric study shown in Fig. 4.6 with no load balancing, 5 original load balancing iterations (LB), and 5 load-balancing-by-dimension (LBD) iterations.

$\sqrt{\text{Num Subsets}}$	f_{reg}	f_{LB}	f_{LBD}
2	1.76	1.76	1.19
3	3.41	2.58	1.49
4	5.83	2.38	1.29
5	9.76	4.25	2.13
6	10.02	3.1	1.33
7	11.17	3.9	1.49
8	12.44	3.59	1.6
9	12.03	3.99	1.96
10	9.76	5.29	1.5

Table 4.2 shows the percent decrease in f with the original load balancing algorithm and the

load-balancing-by-dimension algorithm relative to no load balancing, and the percent decrease in f with the load-balancing-by-dimension algorithm relative to the original load balancing algorithm. There is consistent improvement for both algorithms for all cases, and the improvement both load balancing algorithms relative to no algorithm generally increases as the number of subsets increases. The improvement of load-balancing-by-dimension over load balancing is notable, with a minimum improvement of 32.43% and a maximum improvement of 71.66%.

Table 4.2: The percent decrease of f with the original load balancing algorithm (LB) and the load-balancing-by-dimension algorithm (LBD) relative to no load balancing, and the percent decrease of f with the load-balancing-by-dimension algorithm relative to the original load balancing algorithm.

$\sqrt{\text{Num Subsets}}$	LB v. No LB	LBD v. No LB	LBD v. LB
2	0.0%	32.43%	32.43%
3	24.38%	56.38%	42.33%
4	59.21%	77.9%	45.82%
5	56.48%	78.17%	49.83%
6	69.05%	86.7%	57.01%
7	65.06%	86.66%	61.83%
8	71.17%	87.1%	55.27%
9	66.84%	83.69%	50.81%
10	45.85%	84.65%	71.66%

4.3.2 Parametric study on the Level-2 experiment mesh

The same parametric study was run on an experiment that PDT simulates, the Level-2 experiment, shown in Fig. 4.7. The Level-2 mesh contains a relatively uniform geometry throughout the mesh with one denser feature in the middle of the top boundary. Fig. 4.8 shows the dense region of the Level-2 mesh with 7 regular, load-balanced, and load-balanced-by-dimension cuts. Figure 4.9

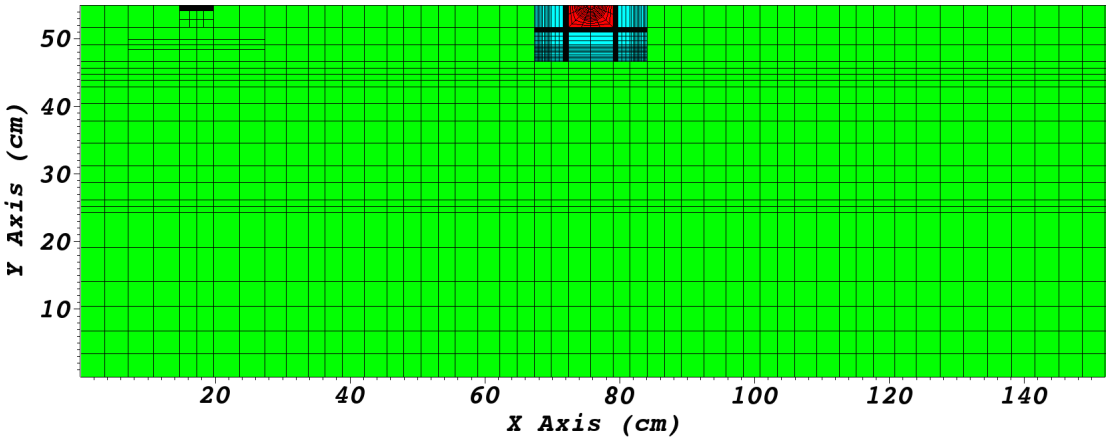


Figure 4.7: The mesh for the Level-2 experiment.

shows the results of this parametric study, which are tabulated in Table 4.3.

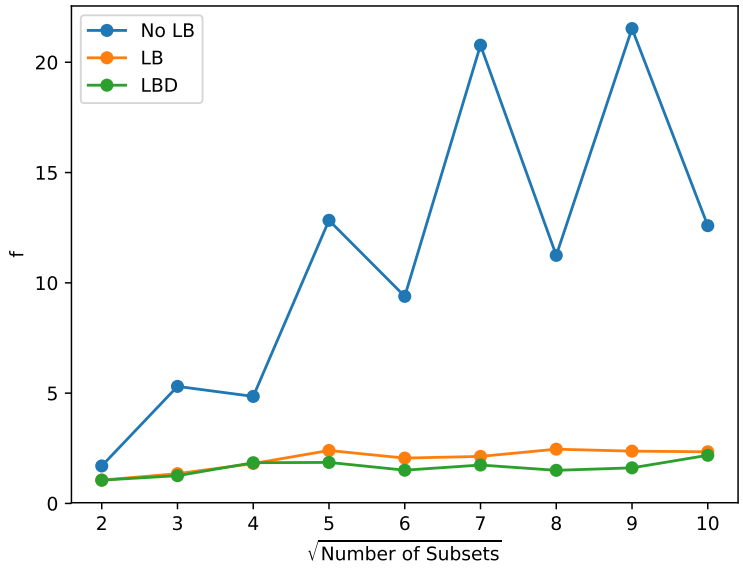


Figure 4.9: The results of the parametric study with no load balancing, 5 original load balancing iterations, and 5 load-balancing-by-dimension iterations for the Level-2 mesh.

With regular cut lines going through the Level-2 mesh, f_{reg} behaves more erratically than the two pin mesh in Fig. 4.1. The natural boundaries are not as uniform, and cells are added in a less

Table 4.3: The tabulated results of the parametric study shown in Fig. 4.9 with no load balancing, 5 original load balancing iterations (LB), and 5 load-balancing-by-dimension (LBD) iterations for the Level-2 mesh.

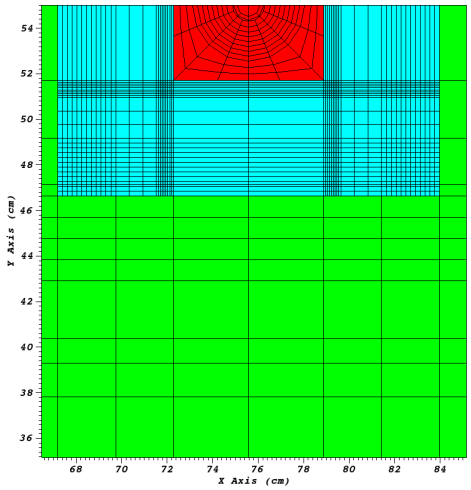
$\sqrt{\textbf{Num Subsets}}$	f_{reg}	f_{LB}	f_{LBD}
2	1.7	1.06	1.06
3	5.31	1.35	1.26
4	4.85	1.81	1.85
5	12.83	2.4	1.86
6	9.39	2.06	1.51
7	20.78	2.13	1.74
8	11.25	2.46	1.5
9	21.53	2.37	1.61
10	12.59	2.34	2.18

consistent fashion. It is noticeable that even numbers of subsets take advantage of the problem's symmetry and consistently have a smaller f_{reg} value.

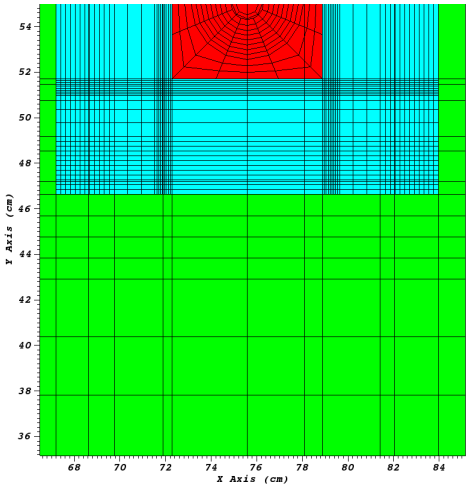
f_{LB} and f_{LBD} behave much less erratically than f_{reg} for the Level-2 mesh. Table 4.4 shows the percent decrease in f with the original load-balancing algorithm and the load-balancing-by-dimension algorithm relative to no load balancing, and the percent decrease in f with the load-balancing-by-dimension algorithm relative to the original load-balancing algorithm. Both algorithms can decrease f up to 90%, but what is notable is the smaller improvement in the load-balancing-by-dimension algorithm relative to the load-balancing-algorithm. With only one dense feature located in the middle of the mesh, moving the y cut lines on a columnar basis is much less advantageous.

Table 4.4: The percent decrease of f with the original load balancing algorithm (LB) and the load-balancing-by-dimension algorithm (LBD) relative to no load balancing, and the percent decrease of f with the load-balancing-by-dimension algorithm relative to the original load balancing algorithm for the Level-2 mesh.

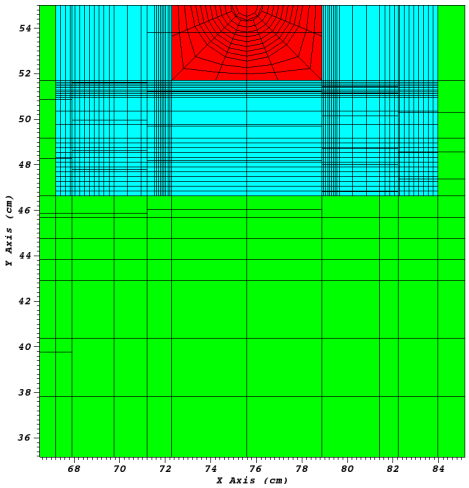
$\sqrt{\textbf{Num Subsets}}$	**LB v. Regular**	**LBD v. Regular**	**LBD v. LB**
2	37.88%	37.88%	0.0%
3	74.56%	76.29%	6.8%
4	62.74%	61.95%	-2.11%
5	81.3%	85.51%	22.53%
6	78.11%	83.95%	26.66%
7	89.74%	91.62%	18.33%
8	78.13%	86.65%	38.96%
9	88.99%	92.51%	31.96%
10	81.41%	82.66%	6.71%



(a) No load balancing.



(b) 5 load balancing iterations.



(c) 5 load-balancing-by-dimension iterations.

Figure 4.8: A demonstration of the original load balancing and load-balancing-by-dimension algorithms on the Level-2 mesh partitioned into 7x7 subsets.

The unstructured meshing capability in PDT allows the user to solve a wider variety of problems, without the need to conform a geometry to a logically Cartesian mesh. The original load balancing and load-balancing-by-dimension algorithms allowed for these unstructured problems to be run more efficiently on large numbers of processors. However, PDT's performance model does not account for unstructured meshes or imbalanced partitions, but rather assumes a structured grid with an equivalent amount of cells per processor. In addition, theoretical studies showed that well balanced partitions when using the load-balancing-by-dimension algorithm did not always translate to a better sweep time. These two reasons motivated the development of a time-to-solution estimator that can estimate the sweep time for a problem given a partitioning scheme (P_i, A_i) regardless of mesh type. The time-to-solution estimator is described in Chapter 5.

5. TIME-TO-SOLUTION ESTIMATOR

With the introduction of unstructured meshes and imbalanced partitions in PDT, we need an expansion of the performance model to estimate the sweep time. In addition, before optimization of the partitioning scheme can occur, it is necessary to have an estimation tool that gives the approximate sweep time for a given partitioning scheme. The time-to-solution estimator serves as the objective function that gets optimized, with the partitions serving as the parameter space. This chapter will detail the time-to-solution estimator, and showcase the results of 2D and 3D verification studies.

The time-to-solution estimator is written in Python 3. Python was chosen as the language for its powerful graph library, networkx [22]. This library gives us a wide variety of graph mathematics and is easy to use. Before detailing the time-to-solution estimator's methodology, a description of the applicable basic graph theory is necessary.

5.1 Graph Theory Applicable to the Time-to-Solution Estimator

Graph theory is a subset of mathematics that focuses on graphs, or structures containing a set of objects that are connected in a particular manner [23]. These objects are called vertices (or nodes), and are connected by edges. Figure 5.1 shows an undirected graph of 4 nodes and 4 edges. The nodes are a mathematical abstraction that can be used to represent a variety of concepts. In this dissertation, they are used to represent the subsets described in Chapter 4.

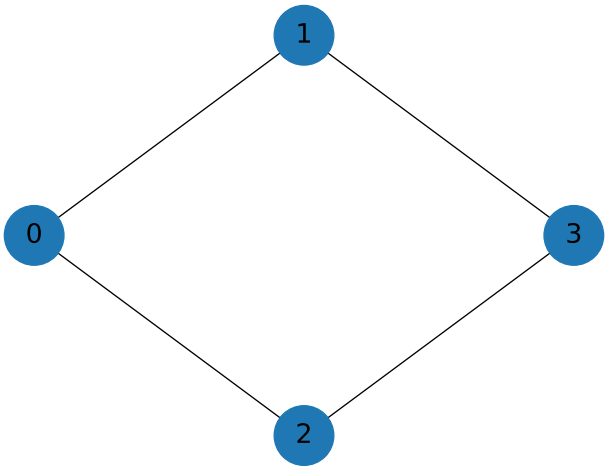


Figure 5.1: An undirected graph with 4 nodes and 4 edges.

Figure 5.1 is referred to as an undirected graph because its edges have no directional information. If we add directional information to the edges of the graph in Fig. 5.1, it becomes a directed graph, shown in Fig. 5.2.

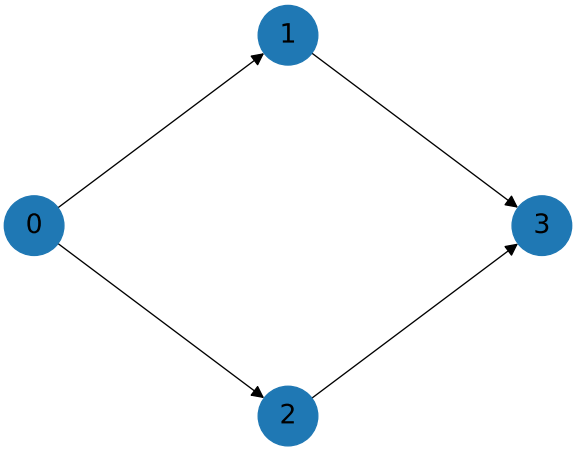


Figure 5.2: A directed graph with 4 nodes and 4 edges.

In this dissertation, we only use Directed Acyclic Graphs (DAGs), or a graph that has no cycles. A cycle is defined as a path on a graph that starts and ends at the same vertex. Figure 5.3 shows a

cycle between nodes 1 and 3.

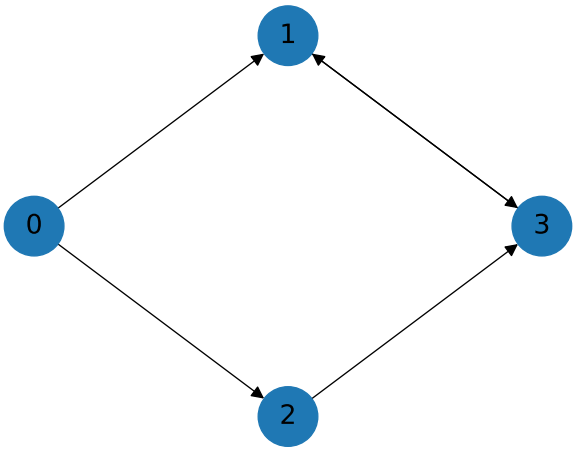


Figure 5.3: A directed graph with a cycle between nodes 1 and 3.

We notice the edge connecting nodes 1 and 3 has a double headed arrow, representing a cycle. Graphs with cycles can be solved for a variety of applications through the use of cycle detection and breaking algorithms, but our partitioning scheme cuts subsets in a fashion that does not allow for cyclical graphs.

Graph edges can be weighted based on the need of the application the graph is being used for. Here, we weight the graph edges to represent the time it takes to solve node A plus the communication time to node B. Figure 5.4 shows a weighted directed graph. In the context of the time-to-solution estimator, subset 1 takes 3 seconds to solve and communicate to subset 3.

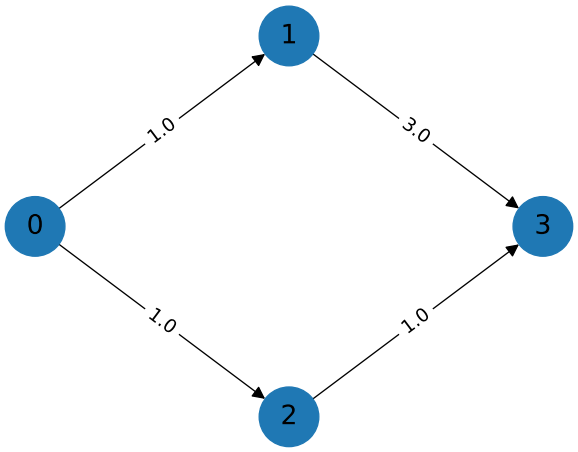


Figure 5.4: A weighted directed acyclic graph with 4 nodes and 4 weighted edges.

The time-to-solution estimator uses Johnson's algorithm [24, 25, 26] to find all shortest paths between all pairs in a weighted directed graph. The weighted shortest path is defined as the path between two nodes that has the smallest weighted sum. For example, the shortest path between nodes 0 and 3 in Fig. 5.4 is $0 \rightarrow 2 \rightarrow 3$.

In our application, we use Johnson's algorithm to assist in calculating the longest path between two nodes in a DAG (needed in Section 5.2.5). The longest path is found by:

1. Multiplying all edge weights by -1,
2. Using Johnson's algorithm to find the shortest (in this case the most negative) paths,
3. Summing the original weights of this shortest path.

Johnson's algorithm is specifically used in this process because it is capable of finding the shortest path even when edge weights are negative.

Now that the applicable graph theory has been reviewed, we detail the methodology of the time-to-solution estimator.

5.2 Method

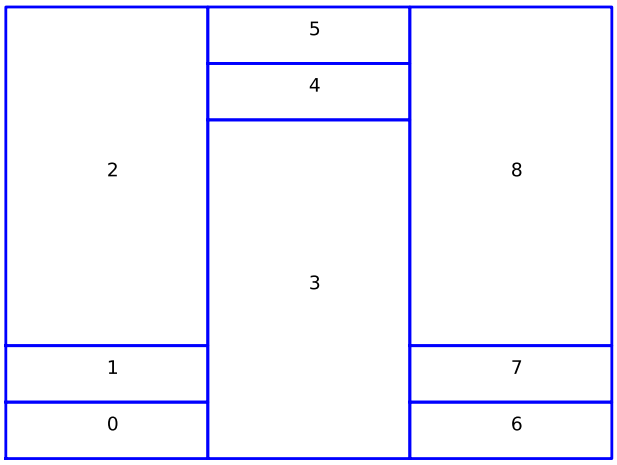
The time-to-solution estimator determines the time to sweep across a domain for a given partitioning scheme by:

1. Building an adjacency matrix,
2. Building Directed Acyclic Graphs (DAGs) from the adjacency matrix, one for each quadrant/octant,
3. Weighting the edges of each graph based on the solve time and communication time of each subset to its neighbors,
4. Adding and modifying graph weights based on how many anglesets are pipelined,
5. Modifying the weights of each graph to operate on the universal timescale,
6. Modifying the weights of each graph to reflect sweep conflicts between octants,
7. Calculating the time to solution.

5.2.1 Building the adjacency matrices

Before building the graph for each quadrant/octant, an adjacency matrix must be built for a given partitioning scheme. The partitioning scheme is the cut lines/planes that partition the mesh into subsets, which correspond to the processor layout. The adjacency matrix provides connectivity information for each subset to its neighboring subset. The adjacency building process relies on a major assumption: the z dimension has partitions all the way across the domain, then the x dimension has partitions per plane, then the y dimension has partitions per plane per column. In future work, these three dimensions can be made interchangeable, but in the present work, this ordering is fixed.

Figure 5.5 shows a partitioning scheme for 3 subsets each in the x and y dimensions with its corresponding adjacency matrix.



$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 5.5: A 3x3 subset partitioning scheme and its corresponding adjacency matrix.

5.2.2 Building the directed acyclic graphs (DAGs)

The adjacency matrices give us directed connectivity information in order to build our graphs. This process differs slightly from 2D to 3D. Both processes use on networkx's DiGraph function to build the DAGs.

5.2.2.1 Building the 2D graphs

In two dimensions, we build four graphs corresponding to four quadrants. We define the quadrants in the following manner:

- Quadrant 0: $\Omega_x > 0, \Omega_y > 0$
- Quadrant 1: $\Omega_x > 0, \Omega_y < 0$
- Quadrant 2: $\Omega_x < 0, \Omega_y > 0$
- Quadrant 3: $\Omega_x < 0, \Omega_y < 0$

Figure 5.6 illustrates this numbering.

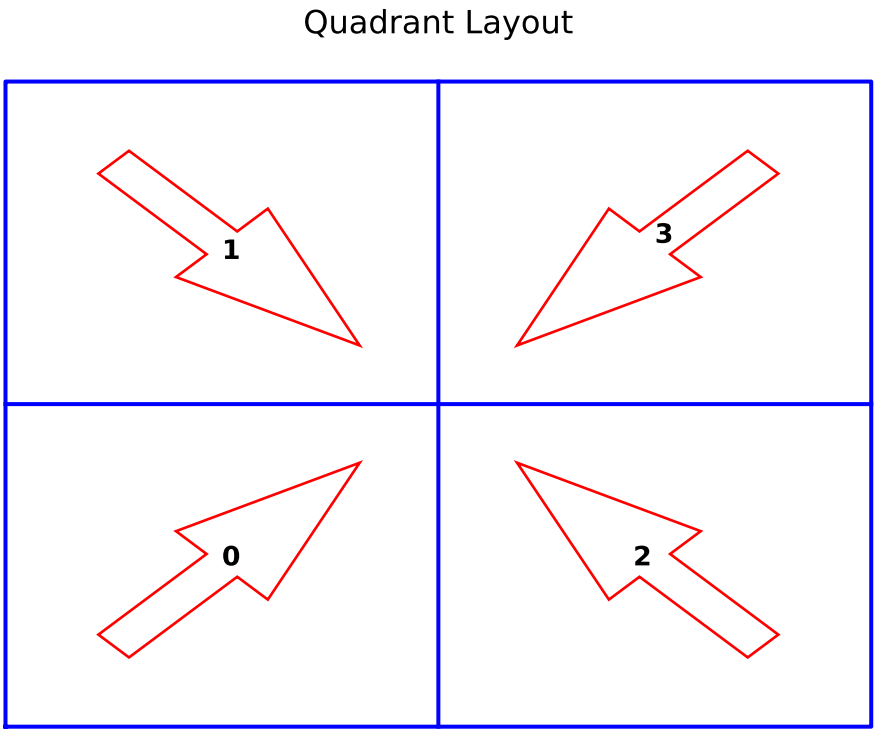


Figure 5.6: The quadrant layout for 2D problems.

The initial adjacency matrix we obtain can be immediately used to build the graphs for quadrants 0 and 3 by using networkx's DiGraph function. We feed the upper triangular portion of the adjacency matrix to DiGraph to get the quadrant 0 graph, and the lower triangular portion to get the quadrant 3 graph. Utilizing the same partitioning scheme shown in Fig. 5.5, pull the upper triangular and lower triangular portions of the matrix, shown in Fig. 5.7.

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 5.7: The upper triangular (left) and lower triangular (right) portions of the adjacency matrix in Fig. 5.5.

Figure 5.8 shows the the DAGS associated with quadrants 0 and 3, built from the adjacency matrices in Fig. 5.7.

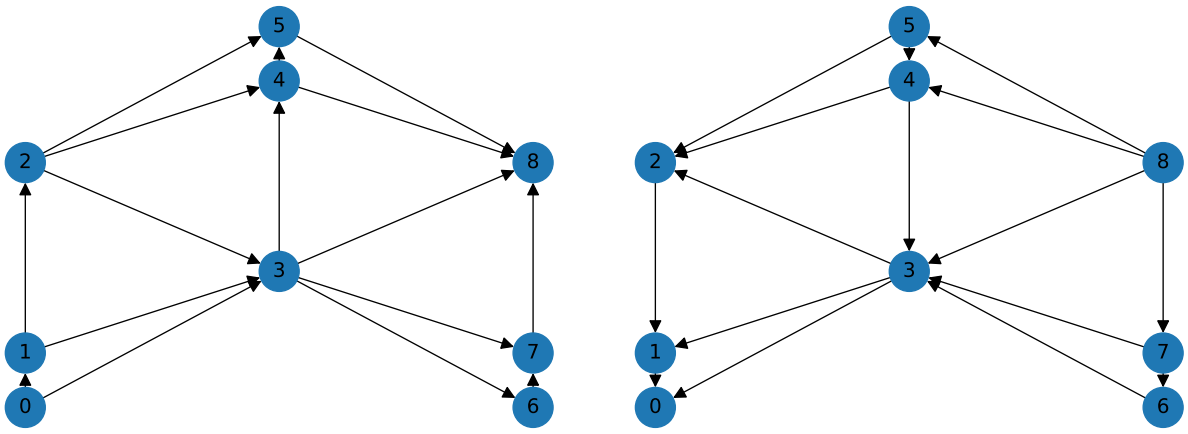
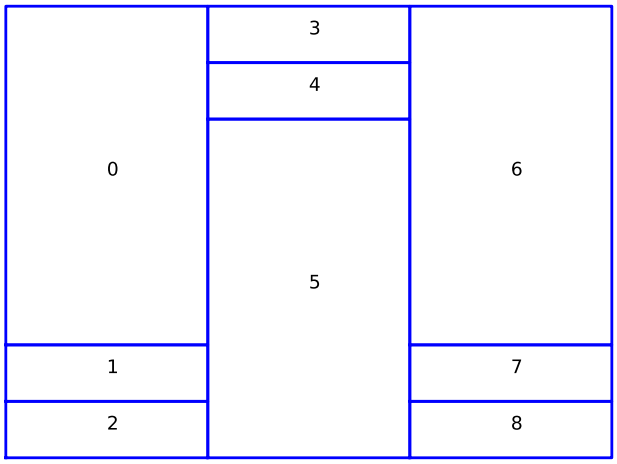


Figure 5.8: The quadrant 0 DAG (left) and the quadrant 3 DAG (right).

Upon inspection of these two DAGs, we see that they show the expected connectivity, dependency, and opposing sweep ordering. Quadrant 0 starts its sweep from subset 0, finishing at subset 8, and quadrant 3 starts its sweep from subset 8, finishing at subset 0.

To obtain the graphs for quadrants 1 and 2, a "flipped" version of the adjacency matrix is necessary. We temporarily renumber the subsets starting from the top left corner, and increasing down each column, as shown in Fig. 5.9.



$$\begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 5.9: The flipped subset ordering and corresponding "flipped" adjacency matrix for the partitioning scheme in Fig. 5.5.

We then get the upper triangular and lower triangular portions of the flipped adjacency matrix to get the connectivity and dependency information for quadrants 1 and 2. We feed the triangular matrices into networkx's DiGraph function, along with a mapping of the flipped subset ids to the original subset ids (shown in Fig. 5.5). Figure 5.10 shows the DAGs for quadrants 1 and 2.

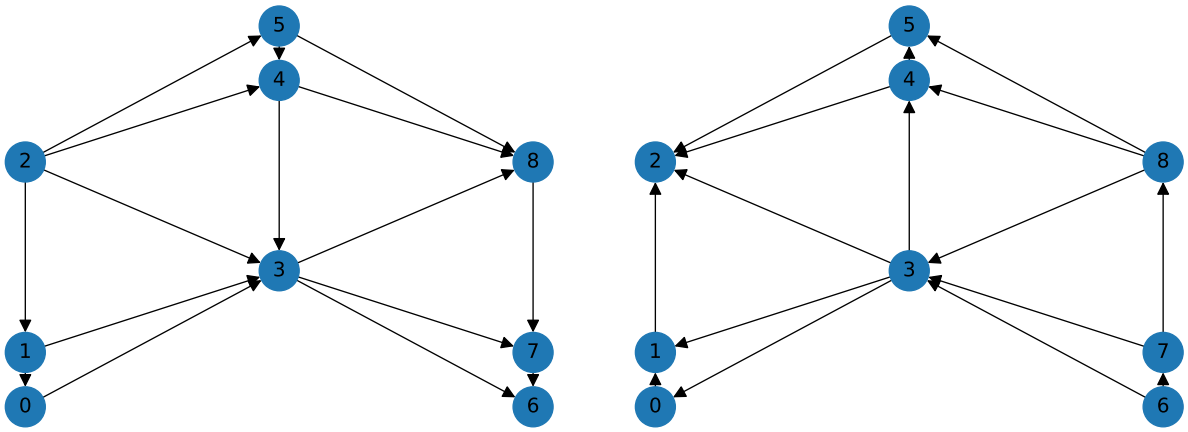


Figure 5.10: The quadrant 1 DAG (left) and the quadrant 2 DAG(right).

Upon inspection of these two DAGs, we see that they show the expected connectivity, dependency, and opposing sweep ordering. Quadrant 1 starts its sweep from subset 2, finishing at subset 6, and quadrant 2 starts its sweep from subset 6, finishing at subset 2.

5.2.3 Weighting the task dependence graphs

Each graph is weighted to reflect the solve time and communication time of each node to its neighbors. Explicitly, the edge weight between node A and node B represents the solve time of node A added to the time it takes to communicate boundary information from nod A to node B. Equation 5.1 shows how the weights are calculated:

$$\text{weight} = \text{mcff} \cdot [T_{wu} + N_n \cdot \text{latency} \cdot M_L + T_{\text{comm}} \cdot N_b \cdot A_m \cdot upbc + \cdot N_c \cdot (T_c + A_m \cdot (T_m + T_g))], \quad (5.1)$$

where:

- mcff = the Multi-Core Fudge Factor, a corrective factor that accounts for performance drop-off from 1 to 8 cores,
- T_{wu} = the time to get into the sweep operator,
- N_n = the number of neighbors this node has to communicate to,

- latency = the machine specific communication latency,
- M_L = the machine specific latency multiplier,
- T_{comm} = the communication time per double,
- N_b = the number of boundary cells shared by node A and node B,
- A_m = the number of angles node A has to solve prior to communicating,
- $upbc$ = the number of boundary unknowns per boundary cell,
- N_c = the number of cells in node A,
- T_c = the time spent solving cell-specific work,
- T_m = the time spent solving angle-specific work,
- T_g = the time spent solving group-specific work.

The weighting function is based on PDT's performance model [11, 12, 13], which is specific to how PDT solves the transport sweep. The cost function can be modified based on different sweep methodologies if a user desires.

As shown in Eq. 5.1, a crucial part of determining the weight of each edge is knowing the number of cells each subset has, and the amount of shared boundary cells with each neighbor. Given a mesh density, the number of cells per subset is given by Eq 5.2:

$$\text{cells per subset} = \int_{x_i}^{x_{i+1}} \int_{y_j}^{y_{j+1}} \int_{z_k}^{z_{k+1}} \text{mesh density } dxdydz, \tag{5.2}$$

where the integral bounds represent the cut plane coordinates that form the subset. Equations

5.3-5.5 calculate the boundary cells along each face in 3D:

$$n_{xy} = \left(\frac{N_c}{V}\right)^{2/3} \cdot L_x \cdot L_y, \tag{5.3}$$

$$n_{xz} = \left(\frac{N_c}{V}\right)^{2/3} \cdot L_x \cdot L_z, \tag{5.4}$$

$$n_{yz} = \left(\frac{N_c}{V}\right)^{2/3} \cdot L_y \cdot L_z, \tag{5.5}$$

where N_c is the number of cells in the subset, V is the subset volume, and L_d is the length of the subset in dimension d. Equations 5.6 and 5.7 show the 2-dimensional equivalents to 5.3-5.5:

$$n_x = \left(\frac{N_c}{A}\right)^{1/2} \cdot L_x, \tag{5.6}$$

$$n_y = \left(\frac{N_c}{A}\right)^{1/2} \cdot L_y, \tag{5.7}$$

where A is the subset area. Equations 5.2-5.7 are exact for structured meshes. However, unstructured meshes do not necessarily have an easily integrable mesh density function to calculate the cells per subset.

5.2.3.1 Determining the cells per subset for unstructured meshes

To get the cells per subset for unstructured meshes we require the vertex data of the mesh, as well as which vertices form each cell in the mesh. With this data, we take advantage of Python's shapely library [27] to determine which cells lie in each subset. From the vertex and cell data of the mesh, we build a lightweight version of each cell as a shapely Polygon.

For each subset we:

1. Loop over the Polygons.
2. For each polygon, check if the Polygon is within the subset. If it is, add it to the cells in that subset.
3. If it is not within the subset, check if the Polygon intersects the subset.
4. If it does intersect the subset, check if it lies on a natural boundary and is outside the subset.

5. If it is not on a natural boundary and truly intersects the subset, then this Polygon's intersection with the subset forms a new cell, and is added to the total number of cells in that subset.

Table 5.1 tabulates the cells count per subset for the time-to-solution estimator and PDT for the mesh in Fig. 4.1 partitioned into 2 subsets in each dimension with regular cuts.

Table 5.1: The cell count per subset for the time-to-solution estimator and PDT for the mesh in Fig. 4.1 partitioned into 2 subsets in each dimension with regular cuts.

Subset	**Cell Count TTS**	**Cell Count PDT**
0	185	185
1	25	25
2	25	25
3	185	185

Figure 5.11 shows the mesh in Fig. 4.1 partitioned into 7 subsets in each dimension with load-balanced-by-dimension cuts. This mesh has many subsets that slice through cells, creating new cells. Figure 5.12 shows the cell counts per subset from PDT and the time-to-solution estimator are in perfect agreement.

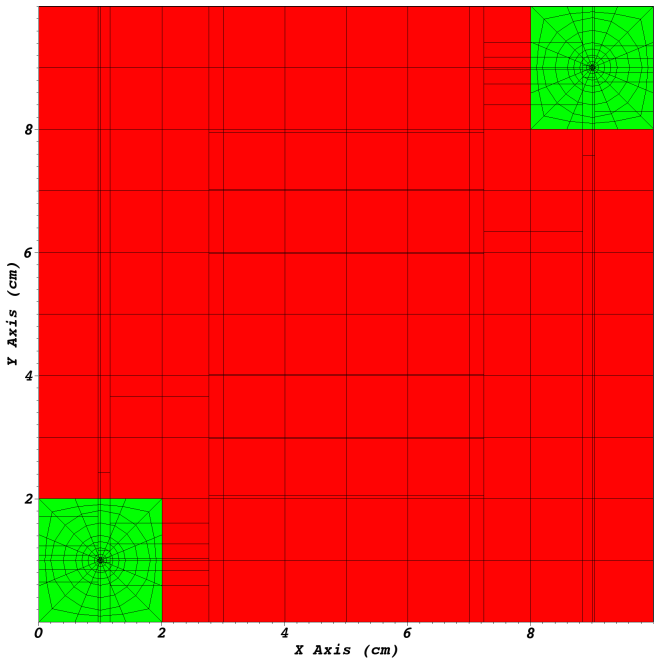


Figure 5.11: The mesh in Fig. 4.1 partitioned into 7 subsets in each dimension with load-balanced-by-dimension cuts. $f = 1.49$

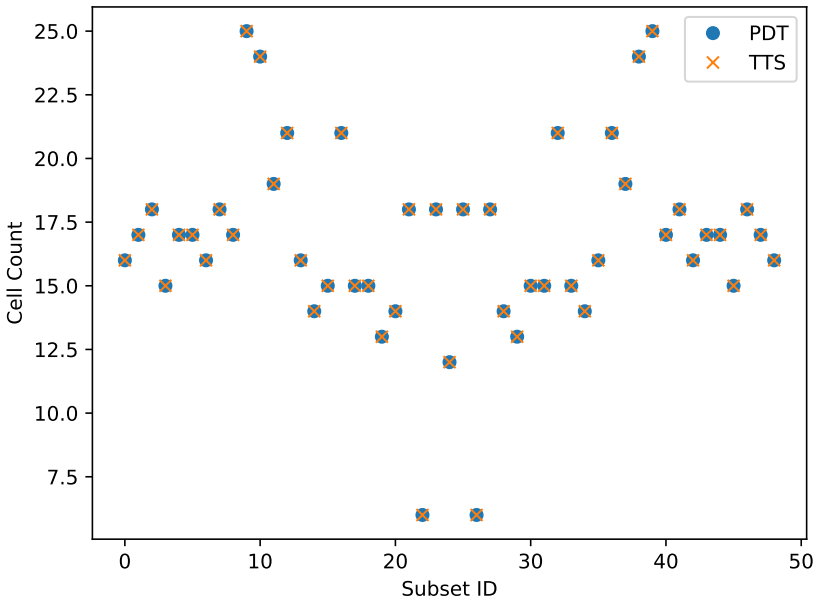


Figure 5.12: The cell count per subset for the mesh in Fig. 5.11 from PDT and the time-to-solution estimator.

With confidence in the cell count per subset in the time-to-solution estimator matching PDT, we make an assumption that the cells in each subset are close to uniformly distributed. This allows us to continue to use Eqs. 5.3-5.7 to calculate the boundary cells in each subset.

With this information, we compute all weights in each graph according to Eq. 5.1. Once graphs are weighted, we add and modify edge weights for the number of angles pipelined per octant/quadrant.

5.2.4 Adding graphs for angular pipelining

If there are angles to be pipelined, the time-to-solution estimator adds a new set of graphs for each additional angle to be pipelined. For example, if there are two anglesets per octant, this would results in 16 graphs, or 1 graph per octant per angleset. Figure 5.13 shows the graphs for two anglesets for a quadrant. A "dummy" node with a value of -2 is added in order to have an incoming edge for the first subset's node. The value of this incoming edge represents when that

angleset starts its sweep.

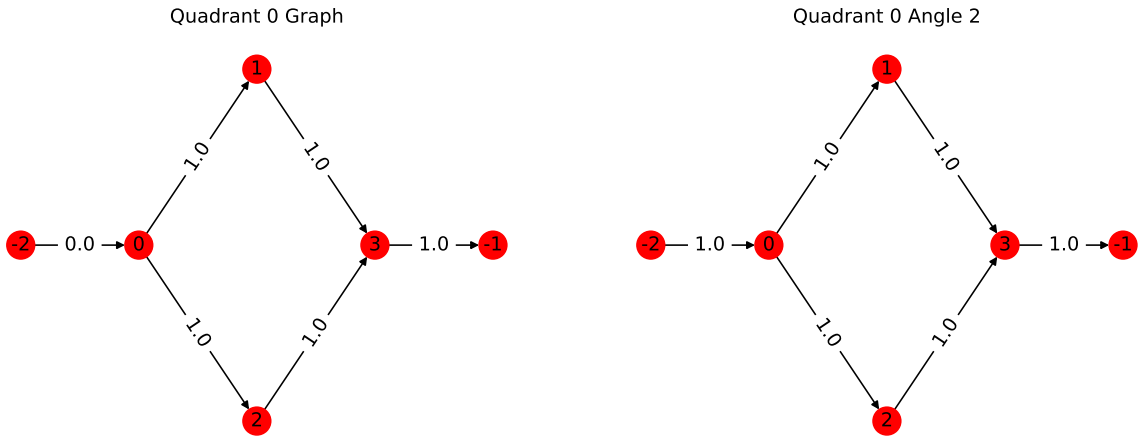


Figure 5.13: The graphs for the first (left) and second (right) anglesets

5.2.5 Modifying the weights of each graph to reflect a universal timescale

Once we have our full set of graphs with angular pipelining accounted for, we set up each graph to reflect a universal timescale, with the goal of knowing when each node in each graph is ready to solve. For each node in each graph, we:

1. Calculate the longest path to the node,
2. Sum the weights of the edges along the longest path,
3. Set all incoming edge values to the node to the sum of the longest path.

The incoming edges to each node in each graph now reflect the time at which a node is ready to solve. This universal edge weighting is used for detecting and resolving conflicts during the sweep. It is important to note that our DAGs are now task dependence graphs (TDGs). We now know which node is ready to solve at each time t, which provides us with a schedule. Figure 5.14 shows a simple example of a DAG becoming a TDG.

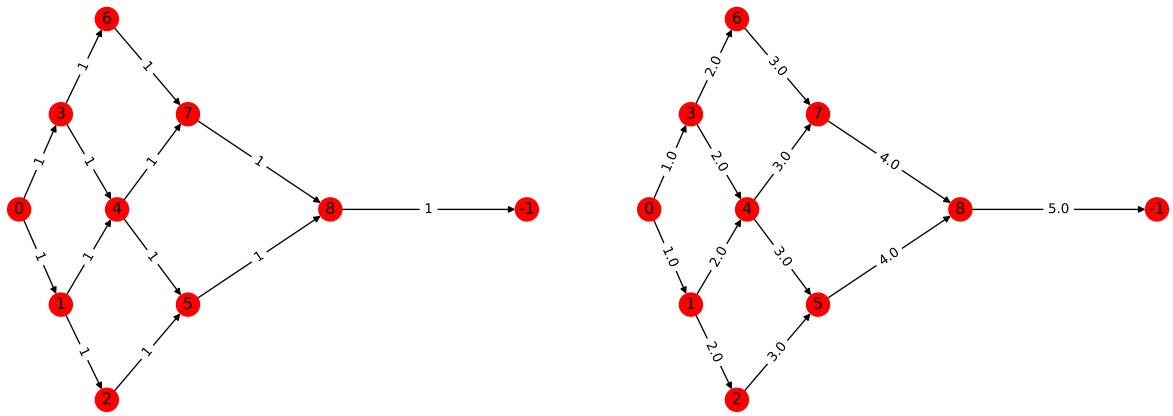


Figure 5.14: A TDG before (left) and after (right) universal edge weighting is applied.

5.2.6 Modifying the weights of each graph to reflect conflict resolution

At this point in the time-to-solution estimation process, we have a graph per octant/quadrant per angleset, with each graph weighted on a universal time scale. For each node in every graph, the incoming edges to the node represent the time t that it is ready to solve at. The time to solution is best summarized as a "marching" process:

1. Set $t = 0$.
2. At time t, find the nodes that are ready to solve or have been solving across all graphs.
3. If at time t, multiple graphs are solving the same node, they are in conflict.
4. The graph that "wins" the conflict does not have its weights modified, while the graph(s) that lose the conflict modify their downstream weights according to how long they are delayed.
5. Update t to the time value of the next node that's ready to solve in any graph.
6. Repeat steps 2-5 until all graphs are finished sweeping.

When a conflict is detected, the time-to-solution estimator uses a first-come-first-serve conflict resolution method. The first graph to arrive to a node will begin solving it, and the remaining

graphs that arrive while it is being solved will incur a delay. The delay is reflected in the remaining graphs by adding the delay as a weight to the applicable edge and all downstream edges in the losing graphs.

If two or more graphs arrive to a node at the same time, the octant with the greater remaining depth-of-graph (simply, more work remaining), wins. In the case of a tie in the depth-of-graph remaining, the graph with the priority direction wins according to the following rules:

1. The graph with $\Omega_x > 0$ wins,
2. If multiple graphs have $\Omega_x > 0$, then the task with $\Omega_y > 0$ wins,
3. If multiple graphs have $\Omega_y > 0$, then the task with $\Omega_z > 0$ wins.

The delay is once again added to the applicable edge's weight and all downstream edges' weights.

5.2.7 Estimating the final time-to-solution

Once all graphs have had their weights modified for conflicts, the graphs now reflect a schedule. The incoming edges to each node in each graph represent what time they are ready to solve. The final weight (the outgoing edge of the final subset) in each graph represents the time it takes for that graph to sweep across its domain. The maximum final weight across all graphs represents the estimate for the time-to-solution for the problem.

5.3 2D Verification

A theoretical study in 2D is run to verify the time-to-solution estimator for 2D partitioning schemes with perfectly balanced partitions. The test problems are verified against a code written by Jean Ragusa that uses a depth-of-graph with an octant priority tie breaker scheduler in two dimensions. The verification study consists of the following problems:

1. 2x2 to 10x10 subsets in x and y with regular partitions and 1 to 6 anglesets per quadrant.
2. 2x2 to 10x10 subsets in x and y with "mildly random" partitions and 1 to 6 anglesets per quadrant.

3. 2x2 to 10x10 subsets in x and y with “random” partitions and 1 to 6 anglesets per quadrant.

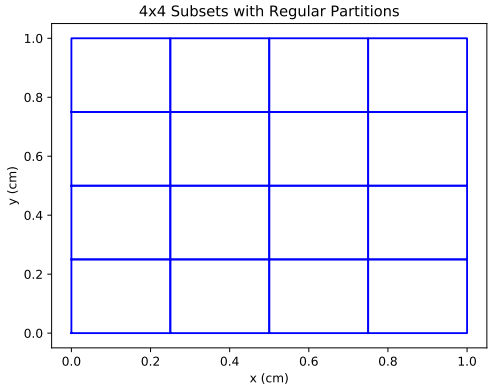
4. 2x2 to 10x10 subsets in x and y with probable worst-case partitions and 1 to 6 anglesets per quadrant.

“Mildly random” partitions keep the cut lines uniformly distributed in x, while the y cut lines vary slightly around the uniformly distributed cut lines of the regular partitions. Figure 5.17 shows examples of this partitioning style. “Random” partitions possesses no such limitations on either set of cut lines, as shown by Fig. 5.19. The “mildly random” and “random” partition styles mimic likely partitioning schemes we can expect from load-balancing-by-dimension.

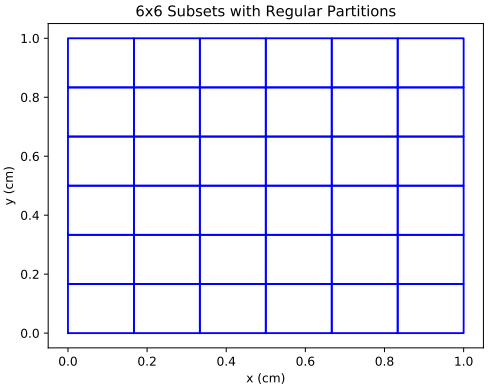
Figures 5.15, 5.17, 5.19, 5.21 show the four partitioning schemes and Figs. 5.16, 5.18, 5.20, 5.22 show the results of the verification study for each partitioning scheme. In the results, a stage is defined as the time it takes to solve all cells in a subset for an angle, and communicate the boundary information to neighboring subsets.

5.3.1 Regular partitions

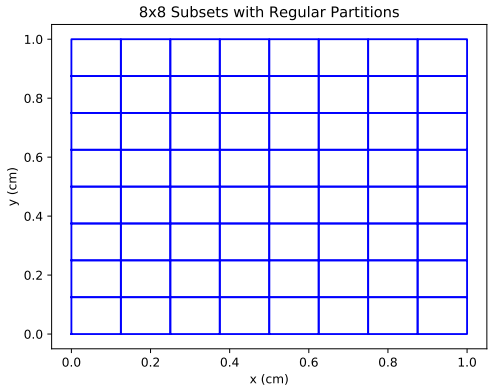
Figure 5.15 shows four examples of the regular partitioning scheme used for the first part of the verification study. Cut lines in both dimensions go all the way across the domain. This reflects the partitioning scheme after the original load balancing algorithm described in Section 4.1 is used.



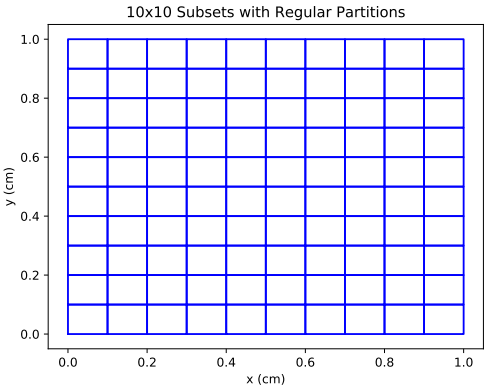
(a) 4x4 subsets with regular partitions.



(b) 6x6 subsets with regular partitions.



(c) 8x8 subsets with regular partitions.



(d) 10x10 subsets with regular partitions.

Figure 5.15: Examples of regular partitioning.

Using regular partitions as shown in Fig. 5.15, the first portion of the 2D verification study was run from 2x2 to 10x10 subsets in x and y and 1 to 6 angles per quadrant. Figure 5.16 shows the results of the time-to-solution estimator (solid line) against Ragusa's code (points) for each test case. The time-to-solution estimator is in perfect agreement for regular partitions with multiple angles per quadrant.

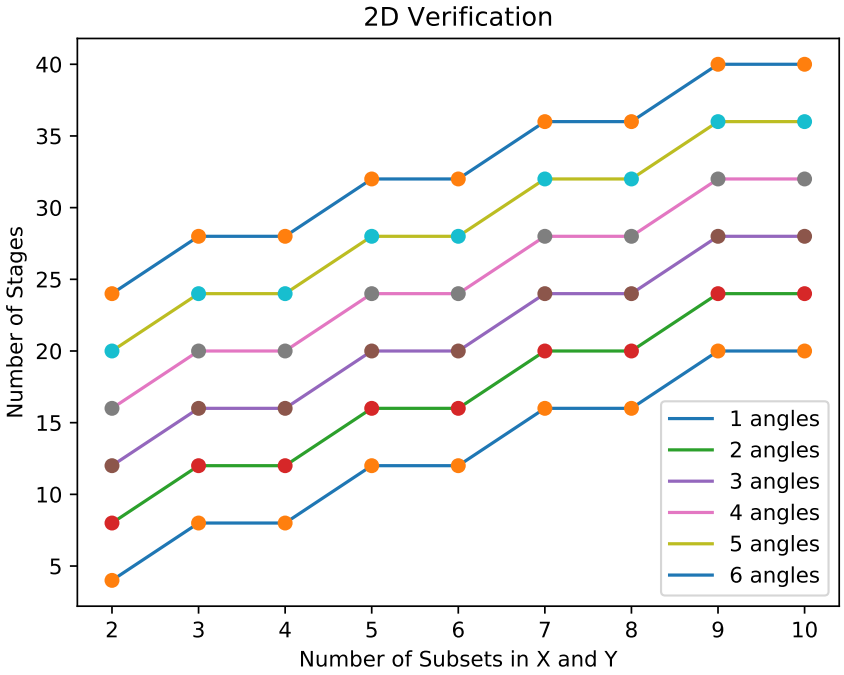
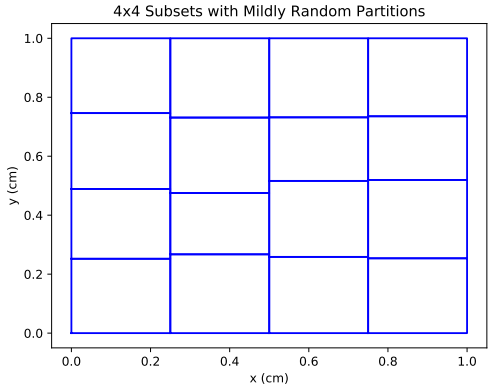


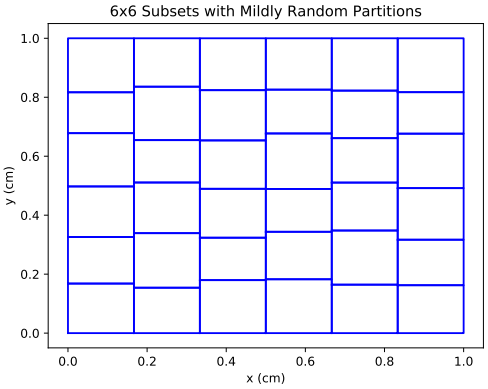
Figure 5.16: A 2D verification suite with regular partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 anglesets per quadrant.

5.3.2 "Mildly random" partitions

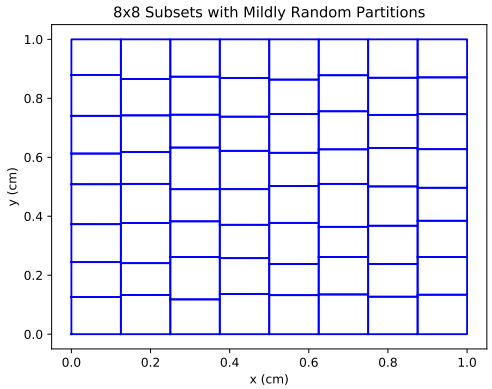
Figure 5.17 shows four examples of the "mildly random" partitioning scheme used for the second part of the verification study. Cut lines in the x dimension go all the way across the domain, and are uniformly distributed. This reflects a possible partitioning scheme after the load balancing by dimension algorithm described in Section 4.2 is used.



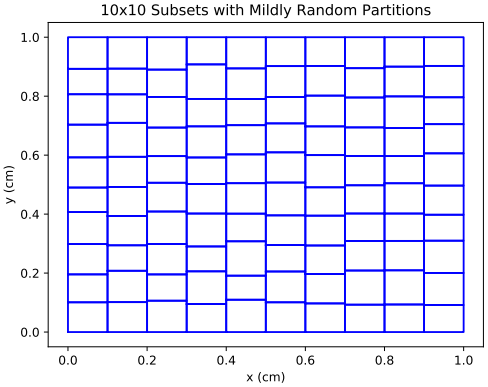
(a) 4x4 subsets with “mildly random” partitions.



(b) 6x6 subsets with “mildly random” partitions.



(c) 8x8 subsets with “mildly random” partitions.



(d) 10x10 subsets with “mildly random” partitions.

Figure 5.17: Examples of “mildly random” partitioning.

Using“mildly random” partitions as shown in Fig. 5.17, the second portion of the 2D verification study was run from 2x2 to 10x10 subsets in x and y and 1 to 6 angles per quadrant. Figure 5.18 shows the results of the time-to-solution estimator (solid line) against Ragusa’s code (points) for each test case. The time-to-solution estimator is in perfect agreement for“mildly random” partitions with multiple angles per quadrant.

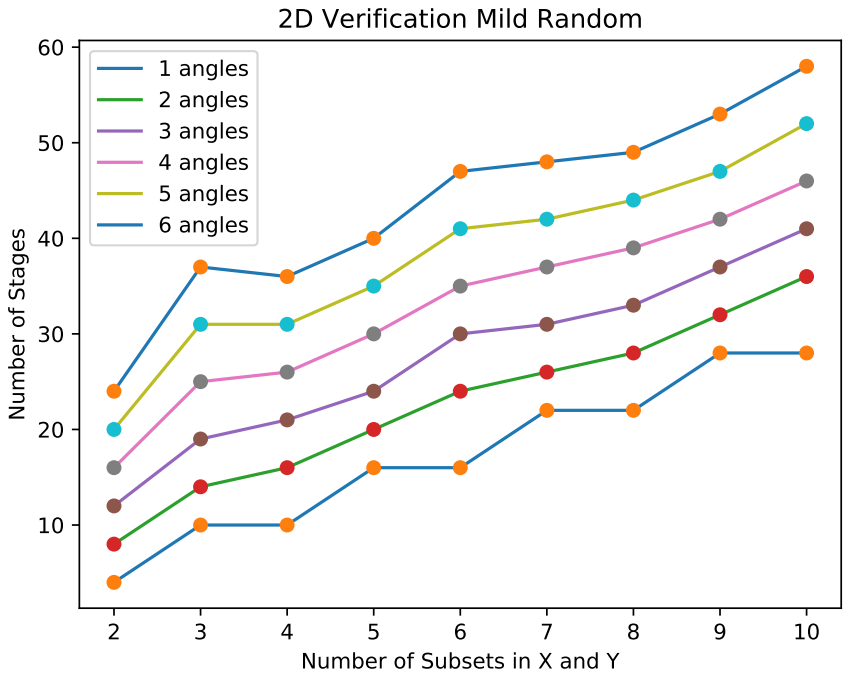
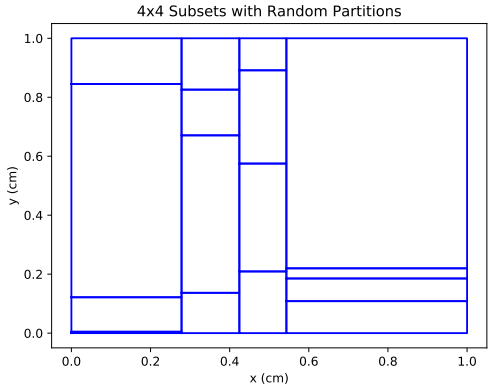


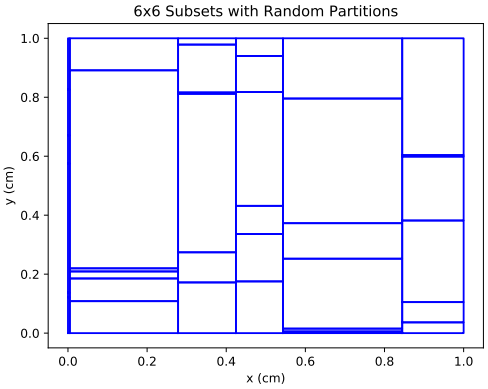
Figure 5.18: A 2D verification suite with "mildly random" partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 anglesets per quadrant.

5.3.3 Random partitions

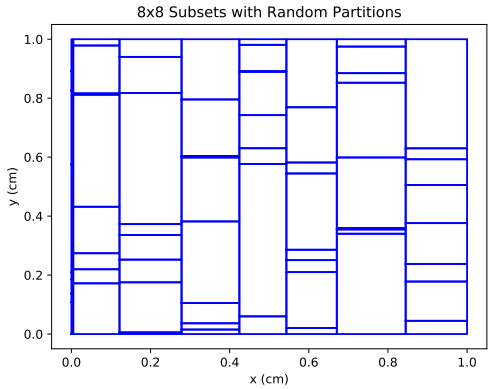
Figure 5.19 shows four examples of the "random" partitioning scheme used for the third part of the verification study. Cut lines in the x dimension go all the way across the domain, but are not necessarily uniformly distributed. The cut lines in y are randomly distributed in each column.This reflects a possible partitioning scheme after the load balancing by dimension algorithm described in Section 4.2 is used.



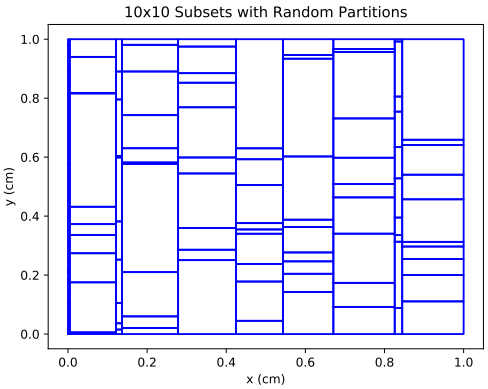
(a) 4x4 subsets with “random” partitions.



(b) 6x6 subsets with “random” partitions.



(c) 8x8 subsets with “random” partitions.



(d) 10x10 subsets with “random”partitions.

Figure 5.19: Examples of “random” partitioning.

Using “random” partitions as shown in Fig. 5.19, the third portion of the 2D verification study was run from 2x2 to 10x10 subsets in x and y and 1 to 6 angles per quadrant. Figure 5.20 shows the results of the time-to-solution estimator (solid line) against Ragusa’s code (points) for each test case. The time-to-solution estimator is in perfect agreement for“random”partitions with multiple angles per quadrant.

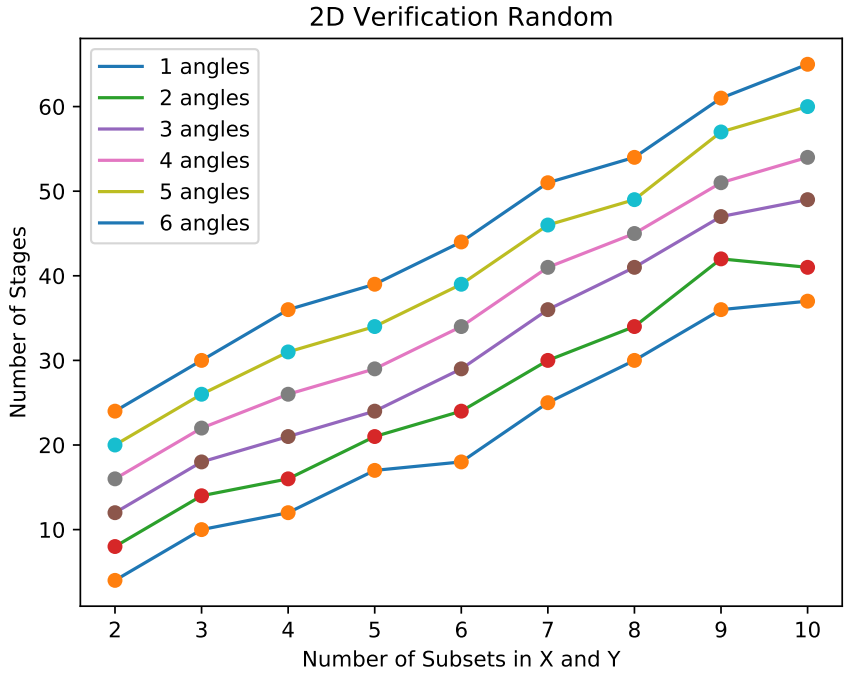
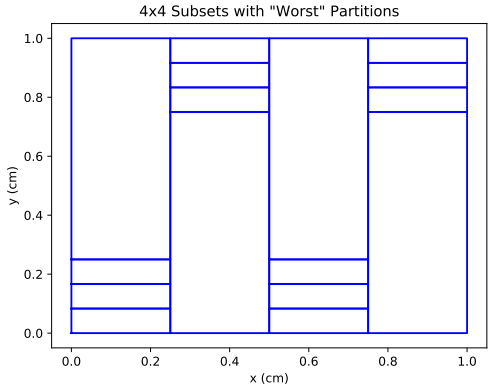


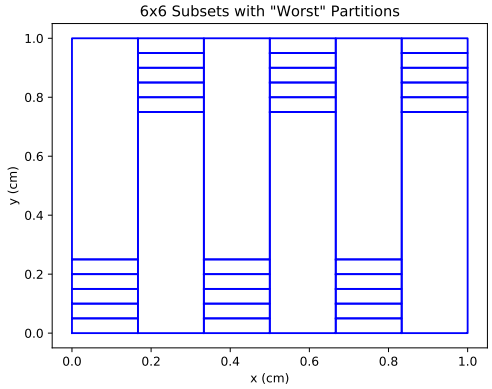
Figure 5.20: A 2D verification suite with "random" partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 anglesets per quadrant.

5.3.4 Probable worst-case partitions

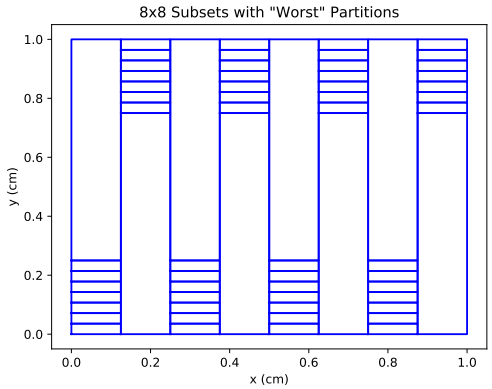
Figure 5.21 shows four examples of the probable worst-case partitioning scheme used for the final part of the verification study. Cut lines in the x dimension go all the way across the domain, and are uniformly distributed. The cut lines in y are distributed on opposing ends of alternating columns.This reflects a possible partitioning scheme after the load balancing by dimension algorithm described in Section 4.2 is used.



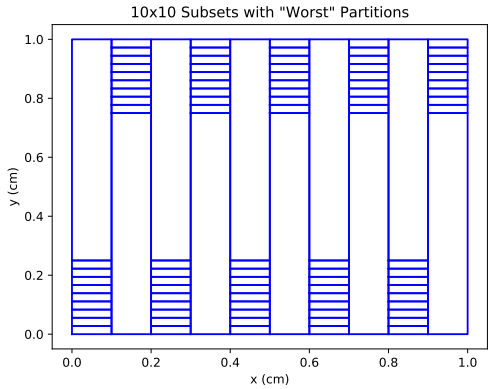
(a) 4x4 subsets with probable worst-case partitions.



(b) 6x6 subsets with probable worst-case partitions.



(c) 8x8 subsets with probable worst-case partitions.



(d) 10x10 subsets with probable worst-case partitions.

Figure 5.21: Examples of probable worst-case partitioning.

Using probable worst-case partitions as shown in Fig. 5.21, the final portion of the 2D verification study was run from 2x2 to 10x10 subsets in x and y and 1 to 6 angles per quadrant. Figure 5.22 shows the results of the time-to-solution estimator (solid line) against Ragusa's code (points) for each test case. The time-to-solution estimator is in perfect agreement for probable worst-case partitions with multiple angles per quadrant.

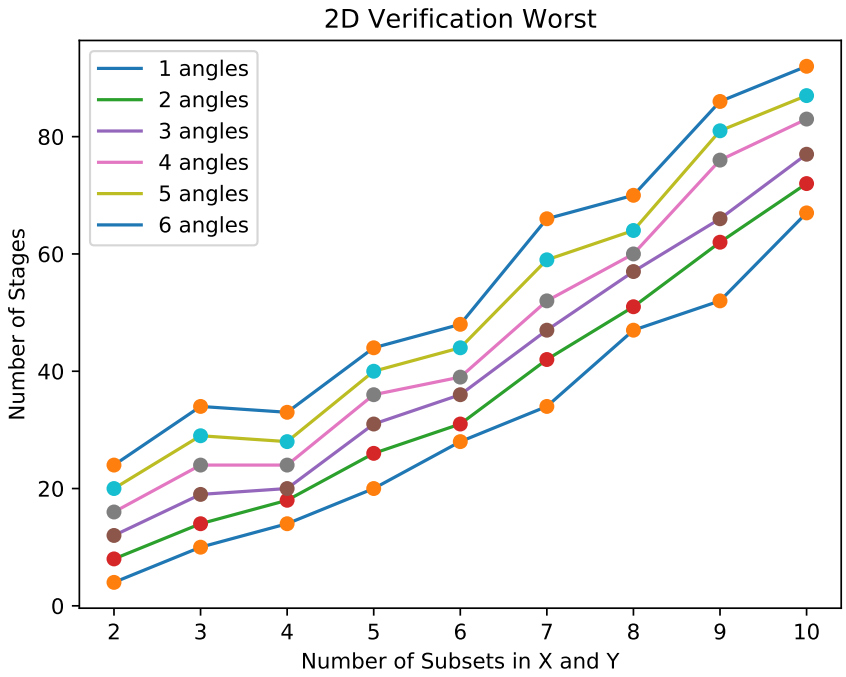


Figure 5.22: A 2D verification suite with probable worst-case partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 angles per quadrant.

5.4 3D Verification

PDT's performance model is used to verify stage counts for 3D problems with regular grids. Figure 5.23 shows PDT's performance model stage counts are in perfect agreement with the time-to-solution estimator's stage counts for 2^3 to 10^3 subsets and from 1 to 6 angles per octant.

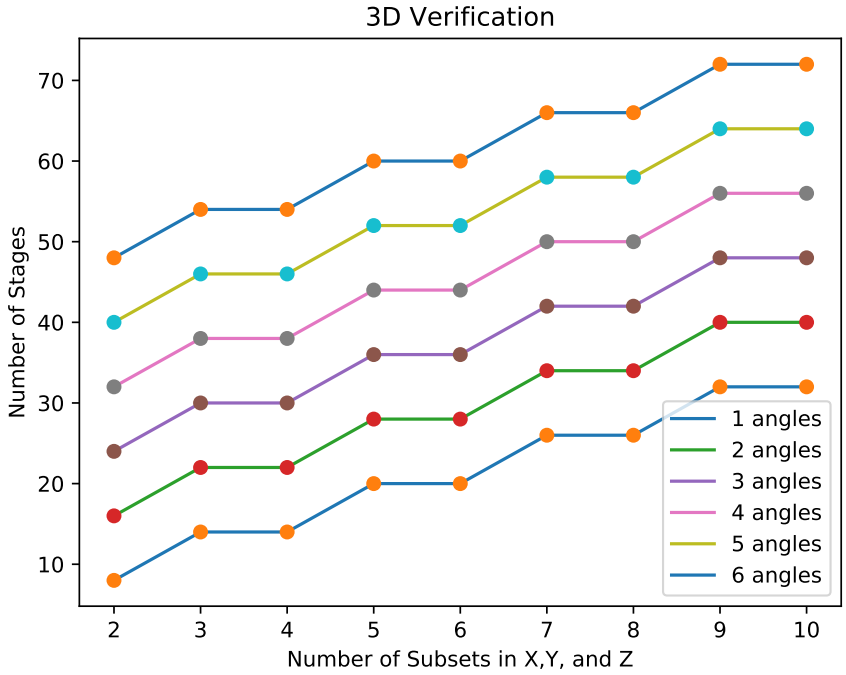


Figure 5.23: A 3D verification suite with regular partitions run from 2^3 to 10^3 subsets with each case being run from 1 to 6 angles per octant.

5.5 PDT's Performance Model vs. Time-to-Solution Estimator

PDT has run a scaling suite out to 90,112 cores on the Quartz supercomputer [28] at Lawrence Livermore National Lab (LLNL). A smaller scaling suite, tabulated in Table 5.2 is the first benchmark case run for the time-to-solution estimator. The suite was run with 1 energy group, 80 directions, $A_m = 10$, $A_x = \frac{N_x}{P_x}$, $A_y = \frac{N_y}{P_y}$, and $A_z = 1$. The following machine parameters were generated and used for the 1 group suite:

- $T_c = 2683.769$ ns
- $T_m = 111.972$ ns
- $T_g = 559.127$ ns
- $T_{\text{byte}} = 4.47$ ns

- latency $= 4110$ ns
- $M_L = 2.5$
- $T_{wu} = 5779.929$ ns
- mcff = 1.32

Table 5.2: The scaling suite parameters ran with 1 energy group, 80 directions, and $A_m = 10$.

Cores	1	8	64	512	1,204	2,048	4,096	8,192	16,384
Nx	16	32	64	128	128	256	256	256	512
Ny	16	32	64	128	128	128	128	256	256
Nz	16	32	64	128	256	256	256	512	512
Px	1	2	8	16	32	32	64	64	128
Py	1	2	4	16	16	32	32	64	64
Pz	1	2	2	2	2	2	2	2	2

Figure 5.24 shows the stage counts of the scaling suite for PDT's performance model and the time-to-solution estimator are in perfect agreement. Figures 5.25 and 5.26 show the time per sweep and parallel efficiency for (1) PDT, (2) the PDT performance model, and (3) the time-to-solution estimator. Table 5.3 tabulates the percent differences for (1) PDT and the performance model, (2) the performance model, and (3) the time-to-solution estimator, and PDT and the time-to-solution estimator.

It is noteable that the time-to-solution estimator consistently returns a smaller sweep time value than PDT's performance model. This is certainly in part due to the assumption from PDT's performance model that each processors at each stage communicate to the same amount of neighbors (three neighbors in 3D, 2 in 2D). In reality, this is not the case, as processors in the corners of the

domain will communicate to fewer neighbors for certain directions, and the final processor for a direction does not communicate to any neighbors (as it has no successors). Because the stage counts for the performance model and time-to-solution estimator are in perfect agreement (as shown in Fig. 5.24), we know that this slight overestimation of the performance model is a reason for its consistently higher sweep times. Even given these differences, the time-to-solution estimator is consistently within 4% of the performance model, as tabulated in Table 5.3.

The performance model's slightly better accuracy in predicting PDT's sweep time is likely due to processor noise and slight variations in latency on Quartz driving up PDT's sweep time. This masks the performance model's slight overestimation of the sweep time.

Figure 5.26 shows PDT's performance model and the time-to-solution estimator scale with near equivalence. The model and the time-to-solution estimator predict PDT's scaling well, although there are slight differences that can be attributed to the differences in sweep times tabulated in Table 5.3.

Table 5.3: The percent difference between (1) PDT and its performance model, (2) PDT's performance model and the time-to-solution estimator, and (3) PDT and the time-to-solution estimator.

Cores	**PDT v. Perf.**	**Perf. v. TTS**	**PDT v. TTS**
1	8.82%	0.68%	9.44%
8	6.67%	1.86%	8.4%
64	2.22%	2.04%	4.22%
512	8.0%	1.52%	9.4%
1024	2.0%	3.67%	5.6%
2048	2.0%	2.21%	4.17%
4096	3.77%	3.24%	6.89%
8192	8.33%	3.8%	11.82%
16384	9.68%	2.7%	12.11%

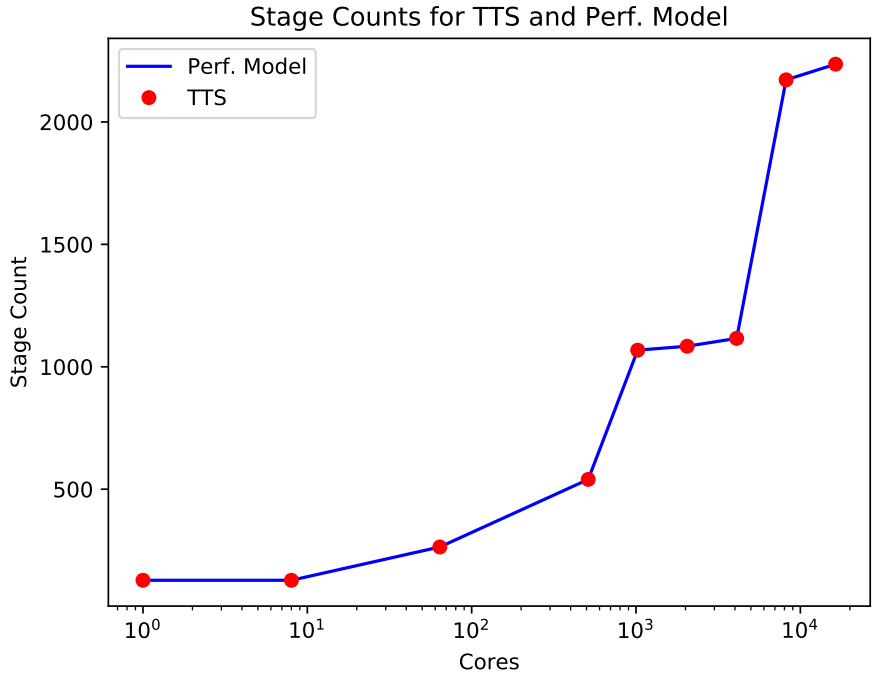


Figure 5.24: The stage counts of PDT's performance model and the time-to-solution estimator for the scaling suite in Table 5.2.

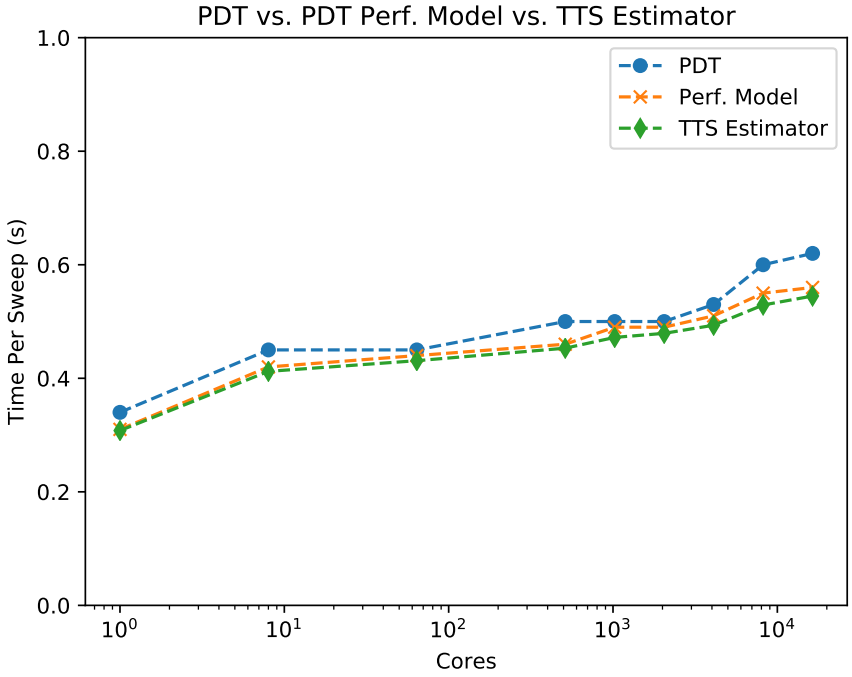


Figure 5.25: The time per sweep of PDT, the PDT performance model, and the time-to-solution estimator.

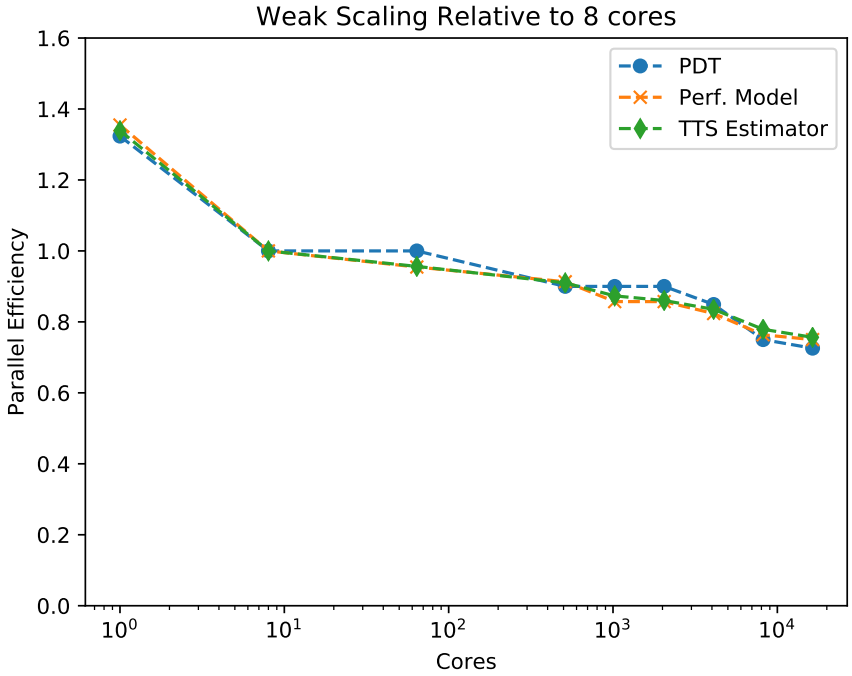


Figure 5.26: The parallel efficiency relative to 8 cores of PDT, the PDT performance model, and the time-to-solution estimator.

5.6 PDT vs. Time-to-Solution Estimator for Unstructured Meshes

The motivation to develop the time-to-solution estimator was born out of the desire to predict sweep time for unstructured meshes. To test how the time-to-solution estimator performs on unstructured problems, we use the meshes from our load balancing parametric study: the unbalanced pin mesh(Fig. 4.1) and the Level-2 experiment mesh (Fig. 4.7).

All problems were run with one energy group, one subset per processor, and 144 angles with $A_m = 36$, or 36 angles per angleset. The machine parameters used for all problems are:

- $T_c = 1208.383$ ns
- $T_m = 65.54614$ ns
- $T_g = 175.0272$ ns
- $T_{\text{byte}} = 4.47$ ns
- latency $= 4110$ ns
- $T_{wu} = 147.0754$ ns
- mcff = 1.181

For each result presented, the problem is run through PDT 10 times on the Quartz supercomputer. This allows us to filter out outlying values due to supercomputer noise by taking the median solve time per sweep for each problem. For each result in PDT, the median sweep time value, the maximum sweep time value, and the minimum sweep time value are plotted to showcase the outlying values that can occur. In order to minimize the effects of the sweep overhead on the timing statistics, the solve time per sweep is calculated from the mean time of ten transport sweeps for each case.

5.6.1 PDT vs. the time-to-solution estimator for the unbalanced pin mesh

Figure 5.27 shows the sweep times for PDT and the time-to-solution estimator for the unbalanced pin mesh with 2 to 10 subsets in each dimension with regular cuts. Table 5.4 tabulates

the percent difference between PDT and the time-to-solution estimator for each of the regular test cases run. We notice that there is better agreement for cases where there are more unknowns. These cases for the the unbalanced pin mesh with regular cuts are cases with 3, 4, 6, 7, 8, and 9 subsets in each dimension, with cut lines cutting through cells and increasing the total number of cells throughout the mesh. Cases with 2, 5, and 10 subsets in each dimension have cuts coinciding with natural boundaries, adding no unknowns to the mesh. More unknowns lead to better timing statistics for PDT, increasing the likelihood of agreement with the time-to-solution estimator.

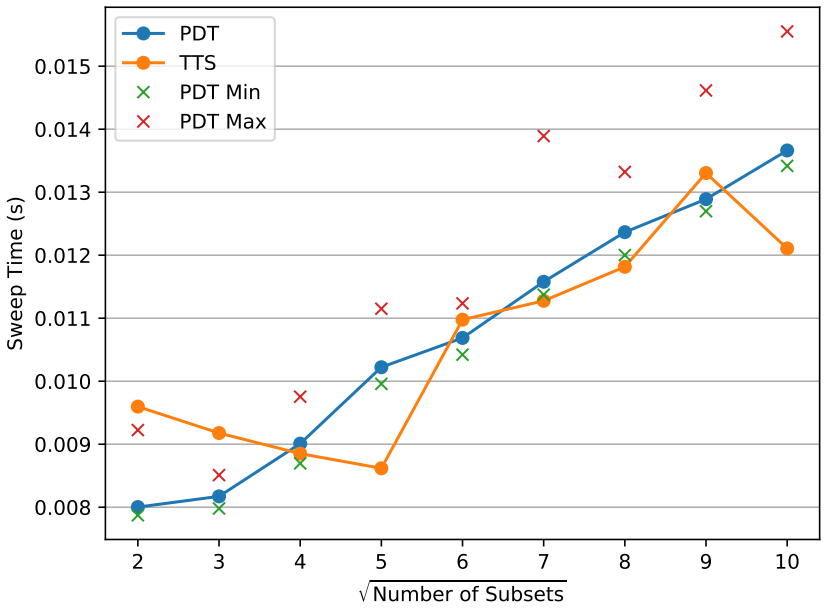


Figure 5.27: The sweep times of the time-to-solution estimator and PDT for the unbalanced pin mesh for 2 to 10 subsets in each dimension with regular cuts.

Figure 5.28 shows the sweep times for PDT and the time-to-solution estimator for 2 to 10 subsets in each dimension with load-balanced cuts. If we focus on the maximum PDT sweep time in the 5 subset case in Fig. 5.28, we can see that PDT can sometimes produce outliers during runs. For that reason, we take the median, not the mean, sweep time value from the ten PDT runs to get the best representation of PDT's solve time per sweep. Table 5.5 tabulates the percent difference

Table 5.4: The percent difference in sweep times between the time-to-solution estimator and PDT for the sweep times shown in Fig. 5.27.

$\sqrt{\text{Num Subsets}}$	PDT vs. TTS
2	19.93%
3	12.3%
4	1.75%
5	15.69%
6	2.71%
7	2.61%
8	4.45%
9	3.21%
10	11.38%

between PDT and the time-to-solution estimator for each case run. We notice that we have poorer agreement with load-balanced cuts than with regular cuts, particularly for lower subset cases. This is possibly due to the first-come-first scheduler in PDT and the schedule of the time-to-solution estimator schedule not being in perfect agreement. While the time-to-solution estimator's schedule is deterministic, there are no guarantees that PDT's first-come-first-serve schedule is repeatable or that it matches the time-to-solution estimator. If there are two tasks that have a similar amount of unknowns, the processor with more unknowns may solve and communicate them faster than the processor with less unknowns. This would cause a disagreement between the time-to-solution estimator and PDT. The low subset cases happen to be the more balanced cases for the unbalanced pin mesh with load-balanced cuts. The 2, 3, and 4 subset cases for the unbalanced pin mesh have load-balance metric values of $f = 1.76, 2.58,$ and 2.38 respectively. With a more even distribution of cells, PDT's first-come-first-serve schedule is likelier to deviate more as certain processors may be faster than their neighboring processors.

Figure 5.29 shows the sweep times for PDT and the time-to-solution estimator for 2 to 10 subsets in each dimension with load-balanced-by-dimension cuts. Table 5.6 tabulates the percent difference between PDT and the time-to-solution estimator for each case run. The load-balanced-by-dimension cases for the unbalanced pin mesh have more consistent agreement than the regular cut and load-balanced cut cases. With load balancing by dimension, more cells are consistently

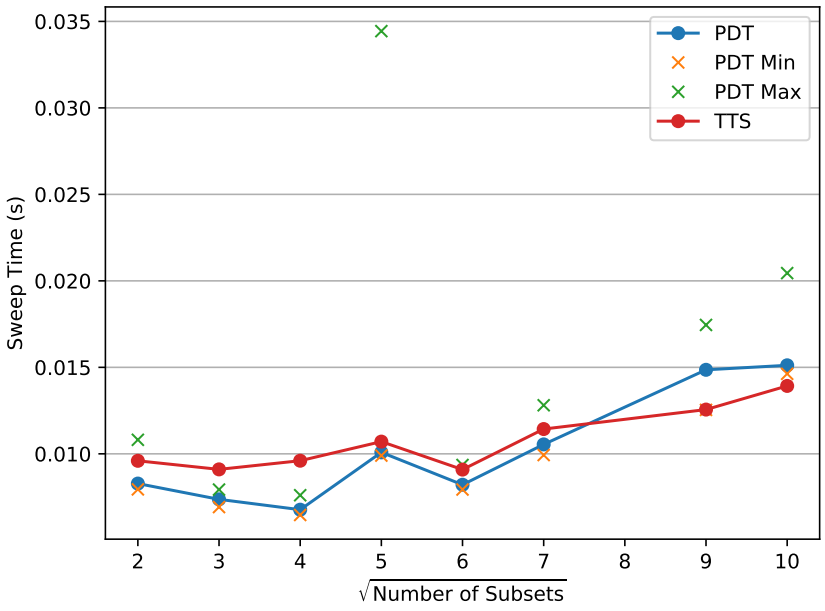


Figure 5.28: The sweep times of the time-to-solution estimator and PDT for the mesh in Fig. 4.1 for 2 to 10 subsets in each dimension with load-balanced cuts.

Table 5.5: The percent difference in sweep times between the time-to-solution estimator and PDT for the sweep times shown in Fig. 5.28.

$\sqrt{\text{Num Subsets}}$	PDT vs. TTS
2	15.9%
3	23.52%
4	41.83%
5	6.15%
6	10.67%
7	8.43%
9	15.47%
10	7.88%

created by slicing through cells in order to get more balanced partitions. By increasing the number of unknowns, we get better timing statistics for the solve time per sweep. In addition, with load-balanced-by-dimension partitions, PDT's first-come-first-serve scheduler is more likely to agree with the time-to-solution estimator's. The communication dependencies inherently created by

load-balanced-by-dimension partitions mitigate the likelihood of faster processors disrupting the expected schedule, as seen with the load-balanced cases.

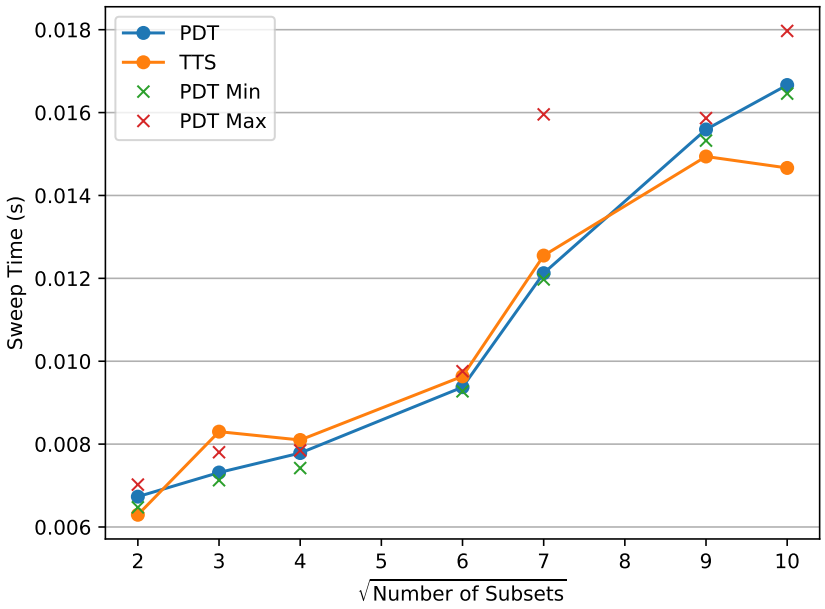


Figure 5.29: The sweep times of the time-to-solution estimator and PDT for the mesh in Fig. 4.1 for 2 to 10 subsets in each dimension with load-balanced-by-dimension cuts.

Table 5.6: The percent difference in sweep times between the time-to-solution estimator and PDT for the sweep times shown in Fig. 5.29.

$\sqrt{\text{Num Subsets}}$	PDT vs. TTS
2	6.5%
3	13.49%
4	4.04%
6	2.78%
7	3.5%
9	4.18%
10	12.01%

Figure 5.30 shows a more refined version of the unbalanced pin mesh, containing 9656 cells as opposed to 420 cells. Figure 5.31 shows the sweep times for PDT and the time-to-solution estimator for the unbalanced pin mesh with 2 to 10 subsets in each dimension with regular cuts. Table 5.7 tabulates the percent difference between PDT and the time-to-solution estimator for each of the regular test cases run.

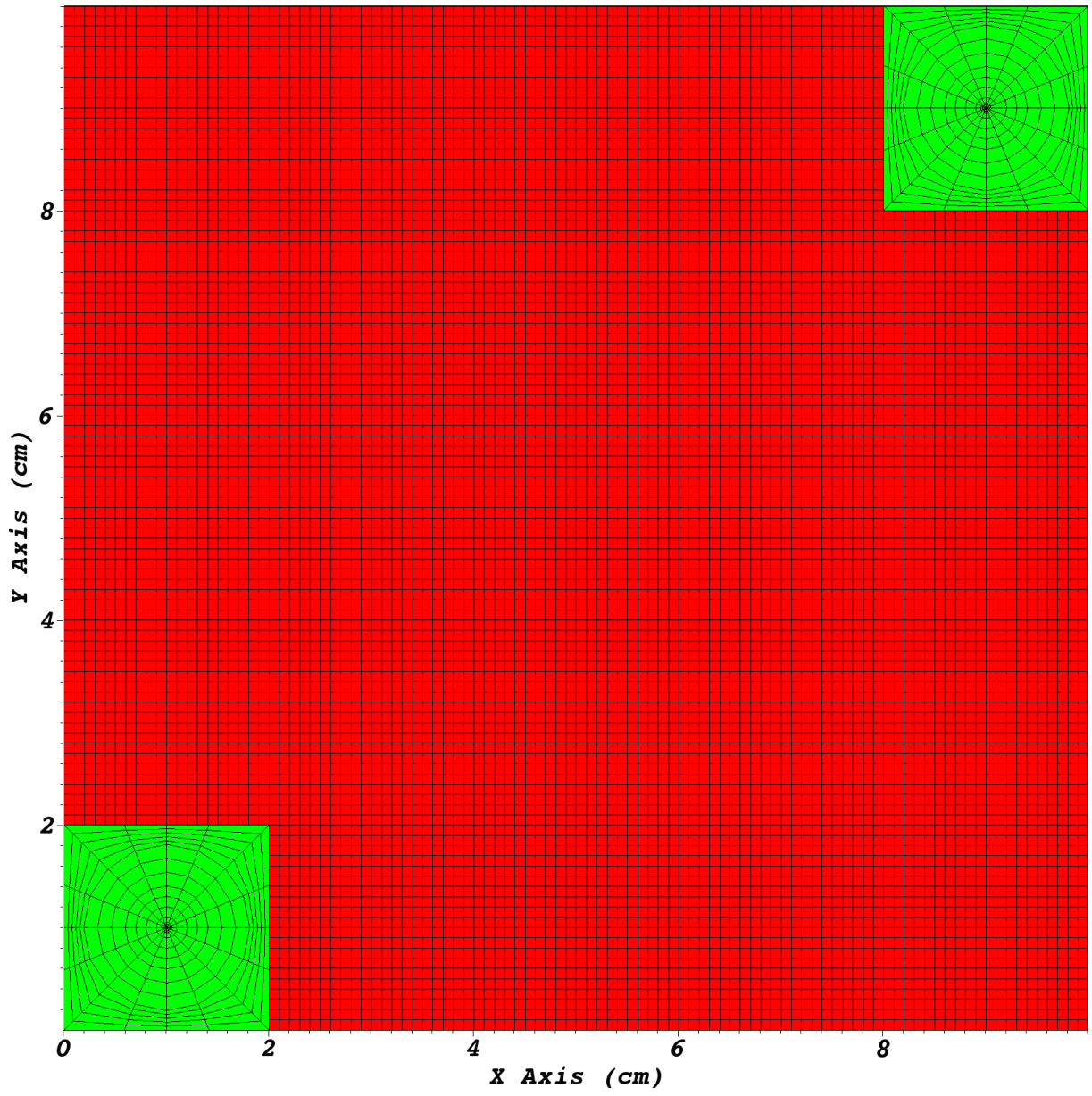


Figure 5.30: A more refined version of the unbalanced pin mesh, containing 9656 cells as opposed to 420 cells.

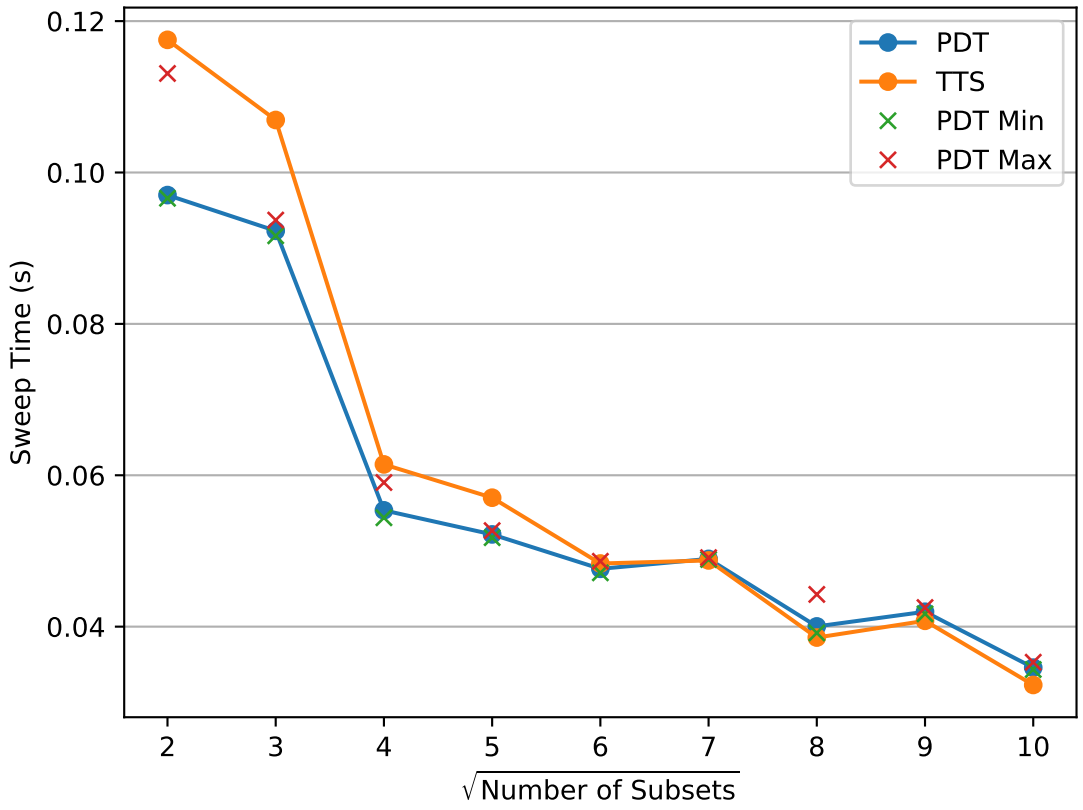


Figure 5.31: The sweep times of the time-to-solution estimator and PDT for the more refined unbalanced pin mesh for 2 to 10 subsets in each dimension with regular cuts

The 9656 cell unbalanced pin mesh time-to-solution estimator sweep times show more consistent agreement with PDT than the 420 cell unbalanced pin mesh. In addition, the behaviors of PDT and the time-to-solution estimator are more consistent with each other with more cells. The low core cases disagree once more, likely due to an overestimation of the latency by the time-to-solution estimator. Properly characterizing the latency for individual core counts is likely to improve agreement at low core counts.

Table 5.7: The percent difference in sweep times between the time-to-solution estimator and PDT for the sweep times shown in Fig. 5.31.

$\sqrt{\text{Num Subsets}}$	PDT vs. TTS
2	21.14%
3	15.87%
4	10.96%
5	9.3%
6	1.47%
7	0.41%
8	3.68%
9	2.85%
10	6.62%

5.6.2 PDT vs. the time-to-solution estimator for the Level-2 experiment mesh

Figures 5.32 and 5.33 show the mesh for the Level-2 experiment with 42 subsets in x, and 13 subsets in y with evenly spaced cut lines and balanced cut lines. The balanced mesh was "hand-balanced" by Marvin Adams, Michael Adams, and Jan Vermaak to achieve better load balancing than the automated load balancing algorithm was able to achieve.

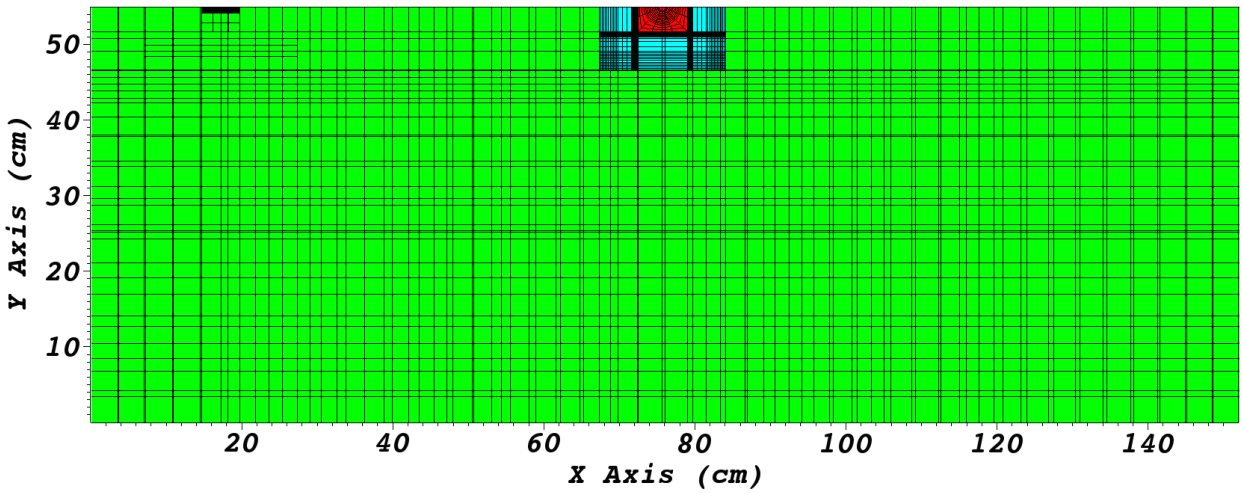


Figure 5.32: The Level-2 experiment mesh evenly partitioned into 42 subsets in x and 13 subsets in y. $f = 32.616$.

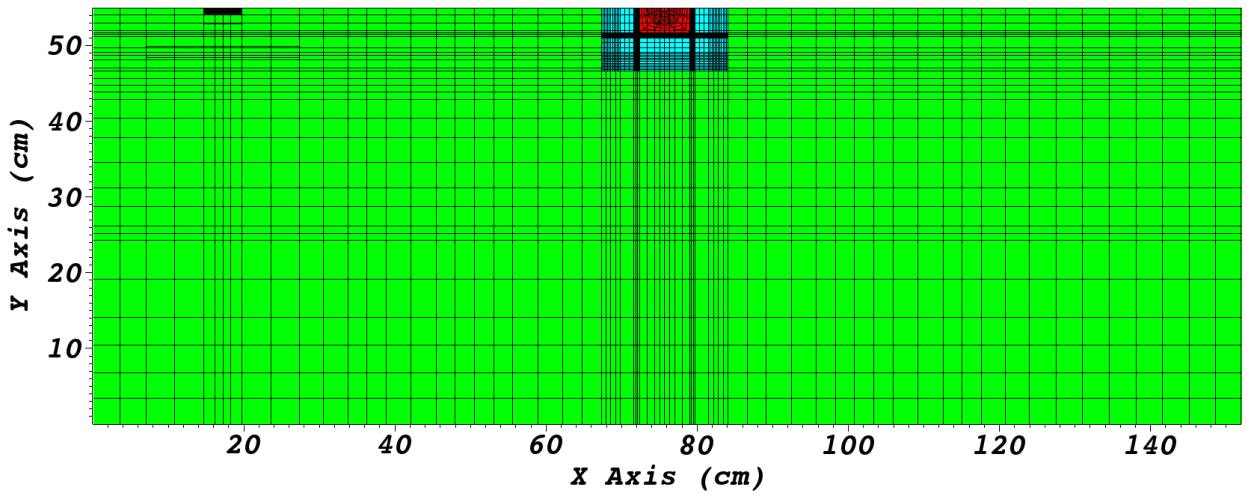


Figure 5.33: The Level-2 experiment mesh partitioned and manually load balanced with 42 subsets in x and 13 subsets in y. $f = 2.386$.

Table 5.8 shows the sweep times for the regular cut and balanced cut Level-2 problems for PDT and the time-to-solution estimator. The minimum and maximum solve time per sweep values are also tabulated. Table 5.9 shows the percent difference between the PDT sweep time and the time-to-solution estimator sweep time. Both the regular and balanced cases have estimated sweep times

Table 5.8: The sweep times for the regular cut and manually balanced cut Level-2 problems for PDT and the time-to-solution estimator.

Case	**PDT (s)**	**TTS (s)**	**PDT Min (s)**	**PDT Max (s)**
Regular	0.07	0.0648	0.0686	0.0889
Manually Balanced	0.0531	0.0535	0.0522	0.0638

Table 5.9: The percent difference for the regular cut and manually balanced cut Level-2 problems between PDT and the time-to-solution estimator.

Case	**PDT vs. TTS**
Regular	7.52%
Manually Balanced	0.63%

within 8% of PDT's sweep times. Because both of these problems have between 3,000 and 4,000 cells, the number of unknowns in the problem can mitigate the effects seen in the unbalanced pin cases. The difference in schedule in addition to latency instabilities at 546 cores are less significant with larger problems.

In the future, the schedule from the time-to-solution estimator should be fed into PDT and see if the agreement improves. In addition, the latency on a machine like Quartz is easy to mischaracterize. A supercomputer of that size and in constant use can have differing latencies depending on how many nodes are in use and which particular nodes are in use. The data generated for the scaling suite was done during dedicated access time, when latencies are much easier to characterize with only one user using the entire machine, explaining the good agreement out to 16,000 cores.

Additionally, the machine parameters were generated empirically using a serial suite of structured mesh problems. In the future, obtaining machine parameters should be attempted using unstructured meshes, and agreement between PDT and the time-to-solution estimator restudied.

6. PARTITIONING OPTIMIZATION

In Chapter 5, we have seen that we can estimate the time-to-solution for a sweep for different partitioning schemes. We use the time-to-solution estimator as the objective function in two optimization methods. The first optimization method utilizes scipy's optimize library, and the second method utilizes knowledge of a problem's mesh layout to assist in partition placement.

6.1 Scipy Optimize

The scipy optimize library [29] provides many tools for optimizing an input function with local and global minimization techniques. Our usage of the optimize library relies on the minimize function, using the basinhopping [30] method as the global optimizer, and the constrained Nelder-Mead method as the local optimizer. We need a global optimization method for larger problem spaces to ensure that cut planes/lines are getting optimized over the entirety of the problem domain, rather than just moving the cut planes/lines close to our initial guess.

The black box tools of scipy optimize are too dependent on the smoothness of the function being optimized. The time-to-solution estimator is not easily differentiable, and therefore not a smooth enough function even for the parameter spaces of a domain decomposed into 3 subsets in each dimension. Although utilizing a tested and documented optimizer would have been ideal, it is clear that we need a method more uniquely suited for our problem.

6.2 CDF Optimization

The "black box" method using scipy optimize's basinhopping and constrained Nelder-Mead minimizers crashes except for very small parameter spaces. However, even with modestly large parameter spaces such as the one seen in the Level-2 experiment (Fig. 6.5), the time-to-solution estimator function is not smooth enough for scipy optimize to honor the constraints or bounds of the problem, leading to the time-to-solution estimator crashing. This lead to the development of an alternative method, the CDF optimization method.

The CDF optimization method utilizes the geometrical information of the problem to attempt

to find optimal cuts. This method prioritizes finding cut line locations that cut along a "natural boundary", and minimizing the total number of times the time-to-solution estimator needs to be run. The time-to-solution estimator for moderately sized problems (such as the Level-2 experiment) can take up to 20 seconds to run for one set of partitions. This rules out a brute force method of running every possible set of partitions. Instead, we select our cut lines from the natural boundaries of the mesh.

A natural boundary is a subset boundary that coincides with the geometrical features of the mesh. In Fig. 6.1, we notice natural boundaries every centimeter in each dimension.

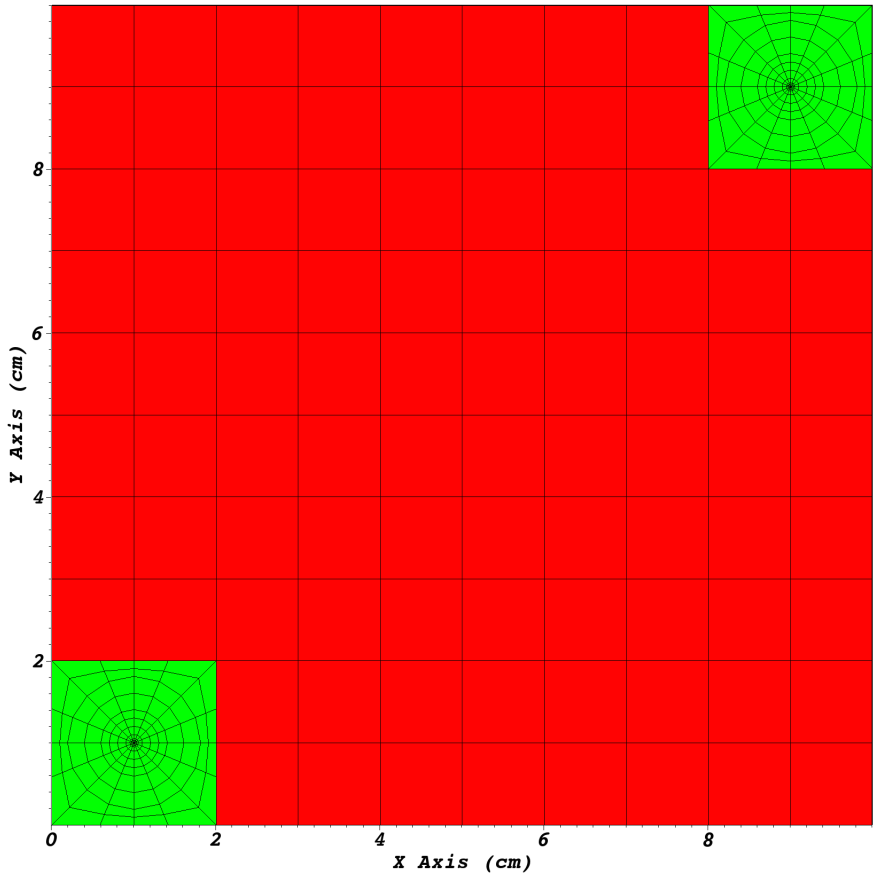


Figure 6.1: An unstructured mesh with natural boundaries at 1 cm intervals in both dimensions.

The CDF optimization method will:

1. Find the most suitable natural boundaries in the x dimension,

2. For each set of columns, find the most suitable natural boundaries in the y dimension,

3. Run all iterations of cut lines selected.

6.2.1 Finding the most suitable natural boundaries

In order to identify natural boundaries, we analyze the detailed cumulative distribution function (CDF) of the vertices in each dimension. The jumps in the CDF correspond to natural boundaries. Figure 6.2 shows the x-vertex CDF of the mesh in Fig. 6.1.

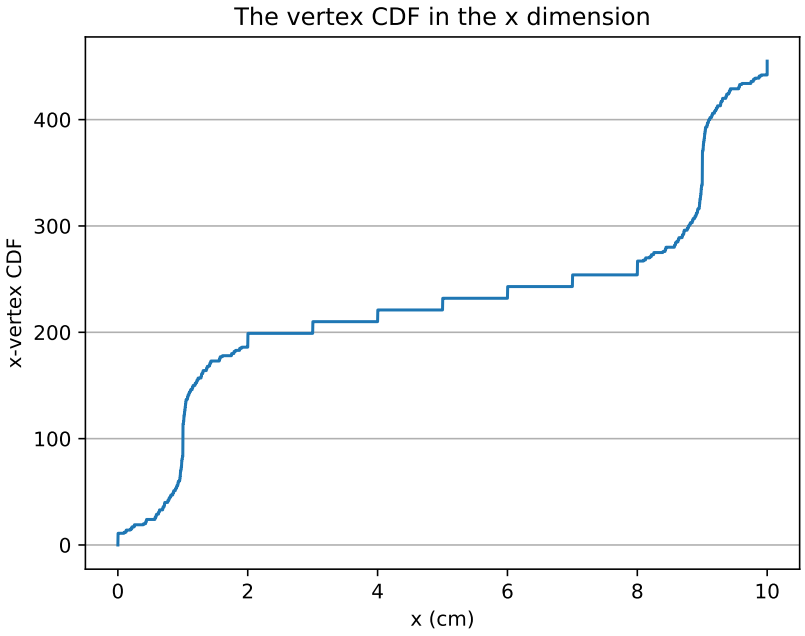


Figure 6.2: The x-vertex CDF of the mesh shown in Fig. 6.1

.

To identify where the jumps in the CDF occur, we take the normalized derivative of the CDF and isolate the largest discontinuities in it. Figure 6.3 plots the derivative of the CDF shown in Fig. 6.2. The largest discontinuities in Fig. 6.3 occur at the instances where there are natural

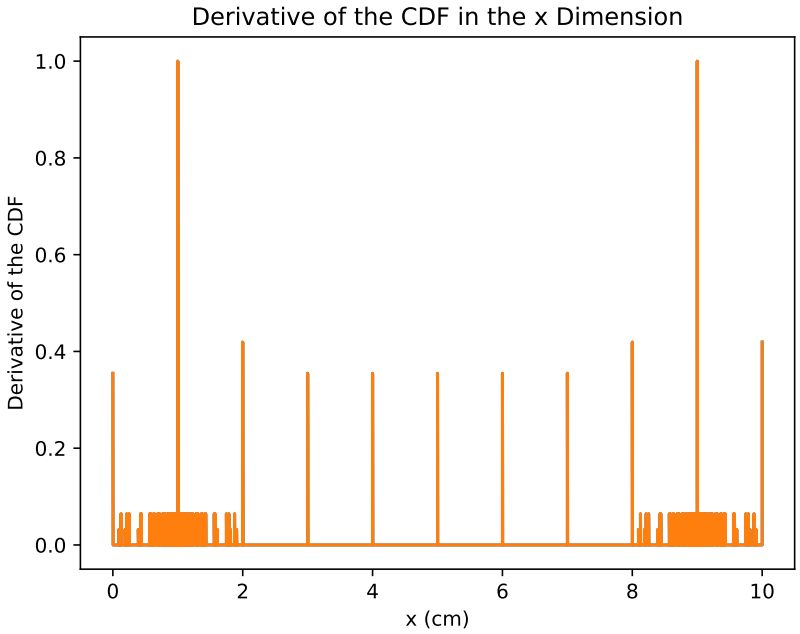


Figure 6.3: The derivative of the CDF shown in Fig. 6.2.

boundaries all the way through the mesh, or at 1 cm intervals. We should note that although the global boundaries of the problem have discontinuities, these discontinuities are obviously not eligible to be chosen as potential partitions.

The process to select the most suitable natural boundaries occurs in two steps. The first step is to to choose cuts that balance the number of vertices per dimension. This is done by finding the intersection of the CDF with an ideal number of vertices per partition, similar to what the redistribution function in the load balancing algorithms does in Fig. 4.3. Instead of using the cells per dimension CDF, we use the vertex per dimension CDF of Fig. 6.2.

Once we have the cuts that balance the vertices of the mesh, we wish to move the cuts to locations that minimize the additions of cells to the mesh. For each balanced cut, we "snap" it to a location corresponding to a natural boundary. We explored choosing where to snap the cut to in one of two ways:

1. Snapping the balanced cuts to the largest discontinuities in the derivative of the CDF.

2. Snapping the balanced cut to a discontinuity that takes into account how close a discontinuity is, as well as the magnitude of the discontinuity.

The first snapping method does not take into account the location of the balanced cuts, and instead just chooses the largest discontinuities in the derivative of the CDF.

The second and third methods choose where to snap each balanced cut based on the distance of the balanced cuts from nearby natural boundaries. Eq. 6.1 shows the calculation of the smallest weighted distance:

$$d = \min_{i \in \text{pool}} \frac{|x^* - x_i|}{J_i} \tag{6.1}$$

where d is the smallest weighted distance, x^* is a balanced cut, x_i is a natural boundary we are testing, and J_i is the magnitude of the normalized derivative corresponding to x_i. The pool is a subset of the discontinuities in the derivative of the CDF. Rather than test against every possible natural boundary, we pull a subset of discontinuities whose magnitudes are over a certain value. This prevents from unnecessarily testing small jumps that only have natural boundaries through a very small portion of the mesh.

6.2.2 Finding natural boundaries for sets of columns

In order to globalize the optimization of the cut lines per column, we set up a binary tree of test cases, such as the one shown in Fig. 6.4. Each layer in the tree represents one set of y cut lines. The first layer, or the root of the tree, represents the case where we try and find the natural boundaries in y throughout all columns, in this case 4 columns. The next layer tries to find two sets of natural boundaries, one set of natural y boundaries through the first two columns, and another set through the final 2 columns. Finally, the last case finds a set of natural y boundaries in each individual column.

Let us consider the mesh of the Level-2 experiment shown in Fig. 6.5. We run this problem with 42 subsets in x and 13 subsets in y. Fig. 6.6 shows 3 stages of choosing partitions for the Level-2 mesh. Fig. 6.6a shows the x partitions, with the y partitions optimized for all 42 columns. This means that we attempted to identify natural boundaries through all 42 columns. Fig. 6.6b

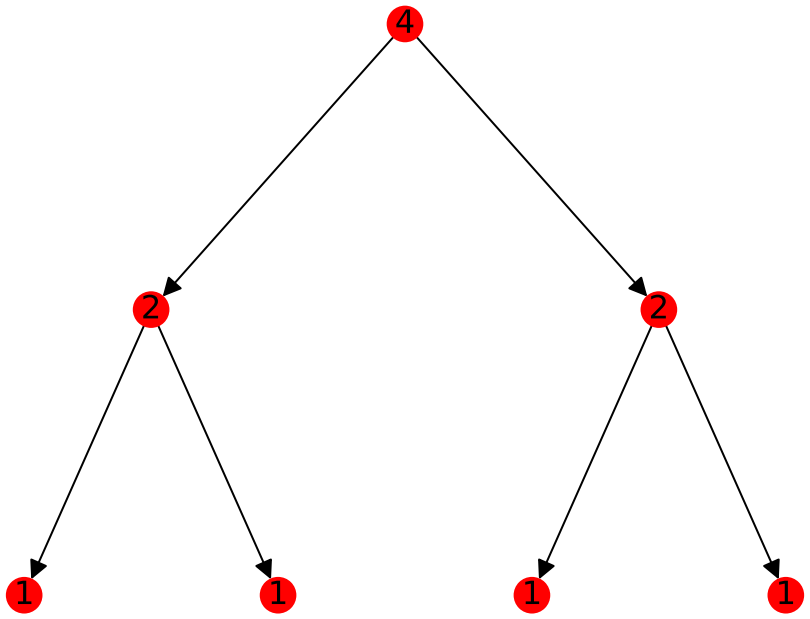


Figure 6.4: A binary tree where each node represents the number of columns we are attempting to find a natural boundary through.

shows the same x partitions, but the y partitions optimized for the first 21 columns independently from the second 21 columns. Fig. 6.6c shows the same x partitions, with the y partitions optimized independently for first 10 columns, then the next 11 columns, then the next 10 columns, then the next 11 columns. We continue this process until we are optimizing y partitions for each column.

This method of optimization allows us to select a set of partitions with an attempt to optimize over the global domain. With this method, we are also not running the time-to-solution estimator a large number of times, instead electing to using use it in an intuitive automation process.

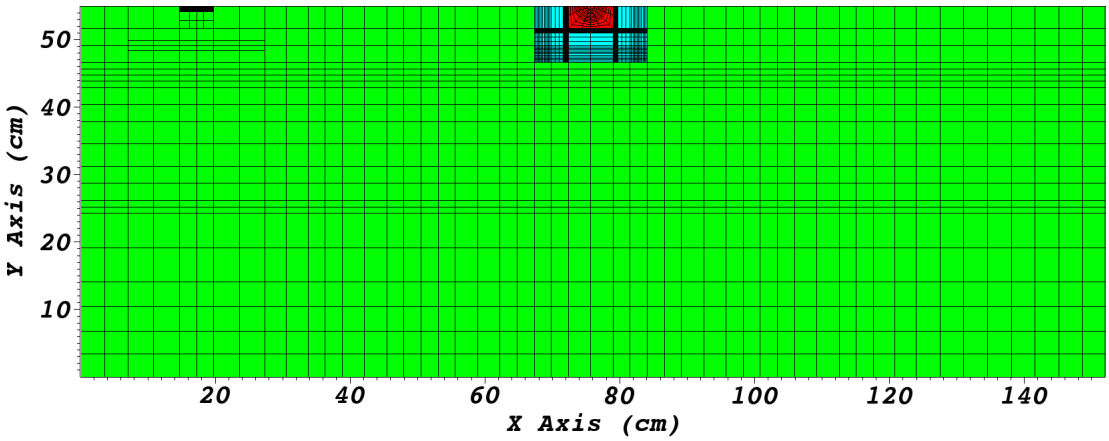
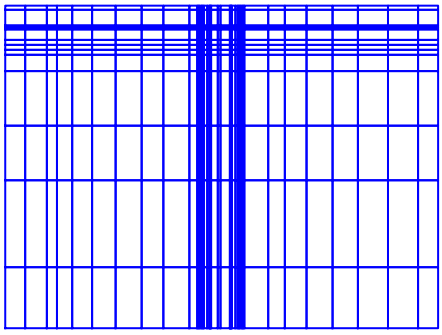
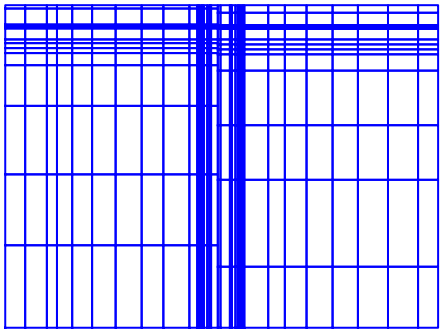


Figure 6.5: The mesh for the Level-2 experiment (same as Fig. 4.7).

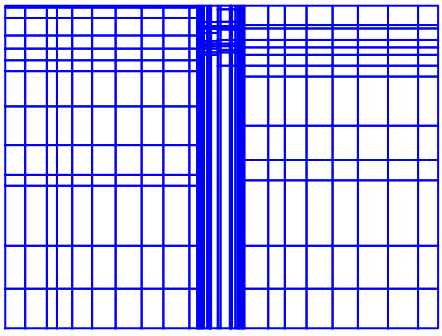


(a) The y-cuts chosen for all 42 columns.



(b) The y-cuts chosen independently for the first 21 columns and the second 21 columns

.



(c) The y-cuts chosen independently for 10,11,10, and 11 columns.

Figure 6.6: The Level-2 Mesh partitions at three stages of the choosing the optimization cut process.

6.3 Optimization Results

The results reported utilize the second "snapping" method, where cuts are snapped to a discontinuity that takes into account how close a discontinuity is, as well as the magnitude of the discontinuity. This method always outperformed the first method of snapping cuts to the largest discontinuities in the derivative of the CDF. For completeness, the results utilizing the first snapping method can be found in Appendix A.

The CDF optimization method was run on the unbalanced pin mesh and the Level-2 experiment mesh. The unbalanced pin mesh was run through the optimization suite from 2 to 10 subsets in each dimension. Figure 6.7 shows the time-to-solution for binary tree, regular, load-balanced and load-balanced-by-dimension cuts on the unbalanced pins mesh from 2 to 10 subsets in each dimension. For low numbers of subsets, the load-balanced-by-dimension partitions outperform

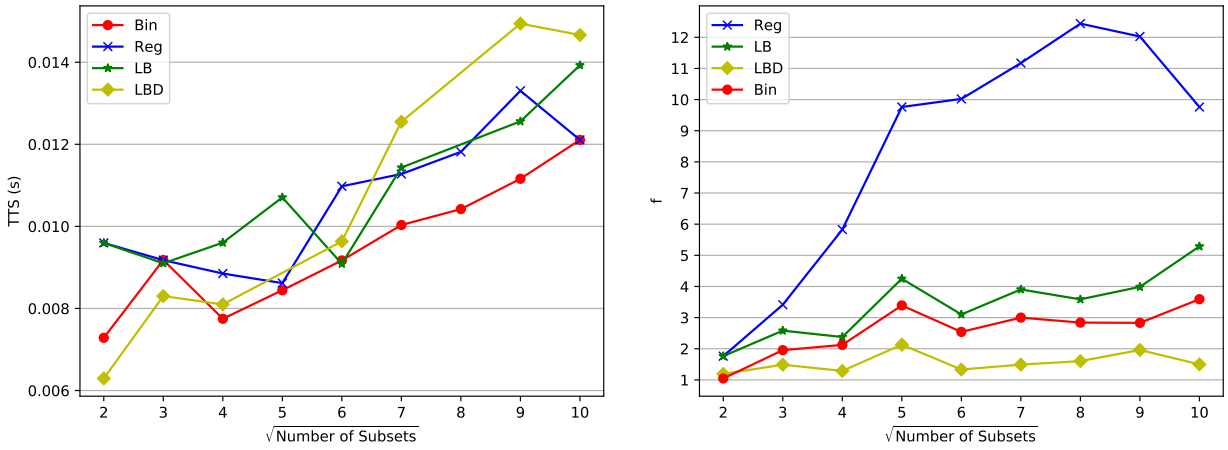


Figure 6.7: The time-to-solution and load-balance metric for binary tree, regular, load-balanced and load-balanced-by-dimension cuts on the unbalanced pins mesh from 2 to 10 subsets in each dimension.

the regular, load-balanced, and optimized partitions. Due to the low number of communications, having partitions that are balanced by column does not incur a large communication penalty, and

having a well-balanced problem is still more important. However, once we exceed 9 subsets, we can see the optimized partitions outperforming the other partition types, with the prioritization of not adding cells paying off.

Figure 6.8 shows the time-to-solution for binary tree, regular, load-balanced and load-balanced-by-dimension cuts on the "heavier" unbalanced pins mesh from 2 to 10 subsets in each dimension. For the majority of cases run with this mesh, the binary tree cuts outperform the regular, load-

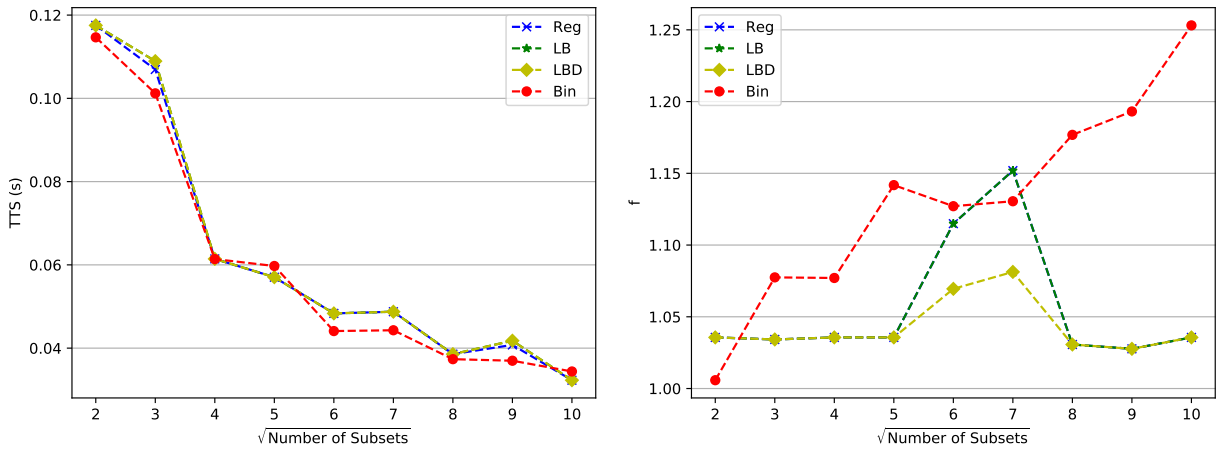
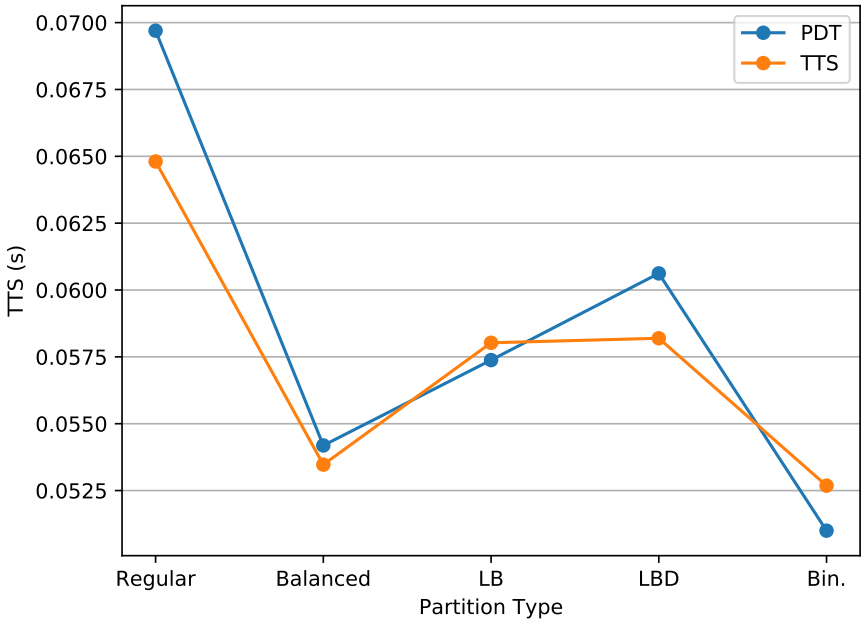


Figure 6.8: The time-to-solution and load-balance metric for binary tree, regular, load-balanced and load-balanced-by-dimension cuts on the "heavier" unbalanced pins mesh from 2 to 10 subsets in each dimension.

balanced, and load-balanced-by-dimension cuts. These cases also further prove that better balance does not necessarily mean a better time-to-solution.

Figure 6.9 shows the sweep times on the Level-2 mesh from the time-to-solution estimator and PDT for (1) regular cuts, (2) hand-balanced cuts, (3) load-balanced cuts, (4) load-balanced-by-dimension cuts, and (5) binary tree cuts. The time to solution of the binary tree cuts is 1.5% better than the hand-balanced cuts. The binary cut suite for the Level-2 experiment mesh took about 2 minutes to find the minimum time to solution, while hand-balancing the mesh took at least an hour. Even though 1.5% may seem like a negligible difference, the difference in time spent to



Type	f
Regular	32.62
Balanced	2.38
LB	5.01
LBD	1.99
Bin.	8.746

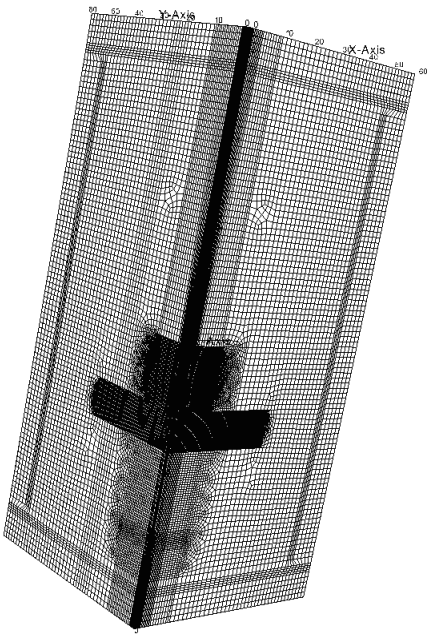
Figure 6.9: The Level-2 mesh sweep times from the time-to-solution estimator for (1) regular cuts, (2) hand-balanced cuts, (3) load-balanced cuts, (4) load-balanced-by-dimension cuts, and (5) optimized cuts. Additionally, the load-balance metric f is reported for each partition type.

get a similar time-to-solution should be taken into account and highlight the significance of the improvement.

Figure 6.10 shows the results of the IM1C experiment mesh run through the time-to-solution estimator for regular, load-balanced-by-dimension, and binary tree cuts with 5 subsets in each dimension. The binary tree cuts beat both the regular and load-balanced-by-dimension cuts by a large margin. In addition, the load-balancing data shows that just because a problem is more balanced, even by a large margin, does not mean the time-to-solution is better.

The CDF optimization method provides a partitioning scheme with a time-to-solution that is better than regular, load-balanced, and load-balanced-by-dimension partitions for the majority of cases run. Choosing cuts that balance the vertices by dimension followed by snapping those cuts to locations less likely to add cells to the mesh proves to be quite effective.

However, we two paths forward for improvement come to mind. With a sped up time-to-



Type	TTS (s)	f
Regular	3.804	12.329
LBD	1.836	1.008
Bin.	0.926	1.918

Figure 6.10: The IM1C experiment mesh run through the time-to-solution estimator for regular, load-balanced-by-dimension, and binary tree cuts with 5 subsets in each dimension. Additionally, the load-balance metric f is reported for each partition type.

solution estimator, we can run several more possible sets of partitions. For example, combinations of natural boundaries could be run rather than choosing the most suitable every time.

Additionally, exploring different formulations for Eq. 6.1, or the calculation of the smallest weighted distance, could yield interesting results. Eq. 6.2 shows a slightly modified version of Eq. 6.1:

$$d = \min_{i \in \text{pool}} \frac{|x^* - x_i|^\alpha}{J_i} \tag{6.2}$$

where α is a parameter that can vary the importance of the distance between a balanced cut and a natural boundary relative to the magnitude. An α value that is less than one would weight the magnitude of the discontinuity in the derivative more heavily, and an α value greater than one would weight the distance between the two cuts more heavily than the magnitude.

7. SUMMARY AND CONCLUSIONS

Transport sweep techniques obtain solutions to the Boltzmann transport equation efficiently by employing discretization techniques that allow for solving the transport equation one cell at a time. Parallelizing the transport sweep offers the ability to solve memory-intensive problems, and with parallel sweep algorithms such as KBA and PDT's hybrid KBA, this can be done efficiently.

However, when unstructured meshes are used, these algorithms can lose effectiveness as unstructured meshes can introduce imbalanced partitions. We combat these imbalanced partitions by load balancing the meshes before sweeping them, attempting to obtain an equivalent amount of work per processor. In Chapter 4, we saw that our two load-balancing algorithms, the original load-balancing algorithm and the load-balancing-by-dimension algorithm, can obtain close to a 90% improvement in the cell count in the heaviest subset for unstructured meshes. However, we noticed that a balanced problem does not signify that it will be swept faster, somewhat defeating the purpose of achieving load balance.

Chapter 5 details the time-to-solution estimator, a tool that predicts the sweep time for a given set of partitioning parameters. PDT has had a performance model that predicted the sweep time for structured, balanced meshes, but had no way to predict the sweep time for imbalanced and staggered (where cut lines don't go all the way through the mesh) partitions. The time-to-solution estimator has no restriction on mesh type, and can be used to predict the sweep time for balanced or imbalanced (importantly, staggered) partitions. The estimator was run through a benchmark suite that mirrored the PDT scaling suite from 1 to 16,384 cores. The time-to-solution estimator was within 4% of PDT's performance model estimator for the sweep time for all problems in the scaling suite, and within 12% of PDT's sweep time for all cases in the scaling suite (shown in Table 5.3). The time-to-solution estimator was also tested against unstructured meshes, with the majority of problems run within 10% of PDT's sweep time.

Chapter 6 describes our method for optimizing partitions. It quickly became apparent that black box tools were not suitable for our problem, due to the time-to-solution estimator not being a

smooth enough function. A secondary method, the CDF optimization method, relies on intuitively interpreting the geometric information in an attempt to find optimal cuts. The detailed vertex CDF is analyzed in each dimension, and jumps in the CDF correspond to locations where natural boundaries (where partition boundaries are the same as mesh boundaries) occur. The CDF optimization method will initially select cuts that balance the mesh, then "snap" those cuts to natural boundaries. This method proved effective at finding optimal partitions for the majority of cases run.

7.1 Future Work

The time-to-solution estimator allowed us to study the sweep time as a function of partitioning parameters. Although we saw good agreement with PDT, particularly for problems with larger numbers of unknowns, we saw some discrepancies. We hypothesize that this is due to the following three reasons:

1. PDT's first-come-first-serve schedule is not consistent with the time-to-solution estimator's schedule,

2. The latency on Quartz is easy to mischaracterize,

3. The machine parameters that are fed to the time-to-solution estimator are generated using structured meshes.

The first reason, the discrepancy between the two schedules, can be remedied by feeding the time-to-solution estimator's schedule directly into PDT and having PDT use it. I believe this would be a worthwhile project for a master's thesis in the future in order to study the effect on the results between PDT and the time-to-solution estimator.

The second reason, the mischaracterization of Quartz's latency, has three potential paths forward. The first is to obtain allocation on a Blue-Gene/Q machine and rerun all problems in this dissertation. Blue-Gene/Q machines tend to have more stable latencies than x86 machines (such as Quartz), and as such can be more easily characterized. Results obtained on a Blue-Gene/Q machine would confirm the x86 noise as a source of discrepancy. The second path forward to this

problem is to attempt to generate latency information for different core counts for x86 machines, rather than have a flat multiplier regardless of core count. The last path forward is to recognize that there is an inherent randomness in the latency on x86 machines, and account for that in the latency term in the time-to-solution estimator.

The third and final reason, the generation of machine parameters, also lends itself to subsequent investigations. Generating the empirical constants used in the time-to-solution estimator should be done with both structured and unstructured meshes to check if the constants' values change significantly. If they do, this could be a significant source of error in the time-to-solution estimator and optimization process.

In addition to the agreement between PDT and the time-to-solution estimator being a significant source of future work, speeding up the time-to-solution estimator would allow for a much larger set of potential partitions to be run through the optimizer. Three paths forward come to mind: rewriting the time-to-solution estimator in a precompiled language rather than an interpreted language, modestly parallelizing the time-to-solution estimator, and machine learning.

Although Python's ease of use and vast number of libraries made it an extremely attractive option for graph algorithms (networkx) and mesh manipulation (shapely), larger problems can take a long time to finish sweeping in the time-to-solution estimator. This can severely hinder our ability to robustly optimize our problems. Our optimization method chooses natural boundaries as the basis for optimizing our partitions, but with a time-to-solution estimator that finishes sweeping in milliseconds rather than tens of seconds, we would be able to run thousands of sets of partitions rather than under 20. If possible, rewriting the time-to-solution estimator in C++ would inherently speed up the code, with the Boost library containing graph and mesh manipulation libraries. Rewriting the time-to-solution estimator in C++ also potentially opens the door to a new set of black box tools that may be effective for functions that are not smooth like the time-to-solution estimator.

Modest levels of parallelism implemented in the time-to-solution estimator could speed up the code enough to drive the time-to-solution estimator's time down significantly. This is not my

personal preferred path forward, as only small portions of the code are embarrassingly parallel, and the Python's parallel overhead could mitigate the advantages.

The field of machine learning lends itself to speeding up the time-to-solution estimation and partitioning optimization processes. With a robust training set, the time-to-solution estimator could be used to construct a model that can quickly predict the time-to-solution for a given set of partitioning parameters. By speeding up the estimation process, we can speed up the optimization process.

REFERENCES

[1] G. Bell and S. Glasstone, *Nuclear Reactor Theory*. Van Nostrand Reinhold, 1970.

[2] K. M. Case and P. F. Zweifel, *Linear Transport Theory*. Addison-Wesley, 1967.

[3] B. Davison, *Nuetron Transport Theory*. Oxford University Press, 1957.

[4] J. Duderstadt and W. Martin, *Transport Theory*. Wiley-Interscience, 1979.

[5] W. L. Dunn and J. K. Shultis, "10 - monte carlo simulation of neutral particle transport," in *Exploring Monte Carlo Methods* (W. L. Dunn and J. K. Shultis, eds.), pp. 269 – 306, Amsterdam: Elsevier, 2012.

[6] J. F. Briesmeister, "MCNP-A general Monte Carlo code for neutron and photon transport," *LA-7396-M*, 1986.

[7] T. M. Pandya, S. R. Johnson, G. G. Davidson, T. M. Evans, and S. P. Hamilton, "SHIFT: A Massively Parallel Monte Carlo Radiation Transport Package," in *ANS M&C2015-Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, Nashville, Tennessee, April 19-23, 2015.

[8] B. Kochunas, B. Collins, D. Jabaay, T. J. Downar, and W. R. Martin, "Overview of development and design of MPACT: Michigan parallel characteristics transport code," in *International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering*, Sun Valley, Idaho, May 5-9, 2013.

[9] R. E. Alcouffe, R. S. Baker, S. A. Turner, and J. A. Dahl, "PARTISN manual," tech. rep., LA-UR-02-5633, Los Alamos National Laboratory, 2002.

[10] Thomas M. Evans et al., "Denovo: A New Three-Dimensional Parallel Discrete Ordinates Code in Scale," *Nuclear Technology*, vol. 171, pp. 171–200, 2010.

[11] Michael P. Adams et al, "Provably Optimal Parallel Transport Sweeps on Regular Grids," in *International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering*, Sun Valley, Idaho, May 5-9, 2013.

[12] Michael P. Adams et al, "Provably Optimal Parallel Transport Sweeps with Non-Contiguous Partitions," in *ANS M&C2015-Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method*, Nashville, Tennessee, April 19-23, 2015.

[13] Michael P. Adams et al, "Provably Optimal Parallel Transport Sweeps on Semi-Structured Grids (Submitted)," *Journal of Computational Physics*, 2019.

[14] Randal S. Baker and Kenneth R. Koch, "An S_n Algorithm for the Massively Parallel CM-200 Computer," *Nuclear Science and Engineering*, vol. 128, March 1998.

[15] T. H. Ghaddar, "Load Balancing Unstructured Meshes for Massively Parallel Transport Sweeps," Master's thesis, Texas A&M University, May 2016.

[16] Tarek H. Ghaddar and Jean C. Ragusa, "An Approach for Load Balancing Massively Parallel Transport Sweeps on Unstructured Grids," in *M&C 2017- International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering*, Jeju, Korea, April 16-20, 2017.

[17] Christian Grossman and Hans-Gorg Roos, *Numerical Treatment of Partial Differential Equations*. Springer, 2007.

[18] W. H. Reed and T. R. Hill, "Triangular Mesh Methods for the Neutron Transport Equation," *Los Alamos National Laboratory Publications*, vol. LA-UR-73-379.

[19] J. O'Rourke, C.-B. Chien, T. Olson, and D. Naddor, "A new linear algorithm for intersecting convex polygons," *Computer Graphics and Image Processing*, vol. 19, no. 4, pp. 384 – 391, 1982.

[20] J. C. Ragusa, "Discontinuous finite element solution of the radiation diffusion equation on arbitrary polygonal meshes and locally adapted quadrilateral grids," *Journal of Computational Physics*, vol. 280, pp. 195–213, 2015.

[21] T. S. Bailey, M. L. Adams, B. Yang, and M. R. Zika, "A Piecewise Linear Discontinuous Finite Element Spatial Discretization of the S_N Transport Equation for Polyhedral Grids in 3D Cartesian Geometry," *Journal of Computational Physics*, vol. 227, pp. 3738–3757, 2008.

[22] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference (Scipy2008)*, pp. 11–15, August 2008.

[23] V. K. Balakrishnan, *Graph Theory*. McGraw-Hill, 1997.

[24] T. H. Cormen, C. E. Leserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press and Mcgraw-Hill, 2001.

[25] P. E. Black, "Johnson's algorithm," *Dictionary of Algorithms and Data Structures [online; accessed 10/16/2019]*, 2004.

[26] D. B. Johnson, "Efficient algorithms for shortest paths in sparse networks," *Journal of the ACM*, no. 24, 1977.

[27] Sean Gillies et al, "Shapely: manipulation and analysis of geometric objects." https://github.com/Toblerity/Shapely, 2007–.

[28] "Quartz." https://hpc.llnl.gov/hardware/platforms/Quartz. Accessed: 10/14/2019.

[29] Eric Jones, Travis Oliphant, Pearu Peterson et. al., "SciPy: Open source scientific tools for Python," 2001–. [Online; accessed 10/16/2019].

[30] D. J. Wales and J. P. K. Doye, "Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms," *The Journal of Physical Chemistry A*, vol. 101, no. 28, pp. 5111–5116, 1997.

APPENDIX A

OPTIMIZATION RESULTS OBTAINED BY SNAPPING CUTS TO THE LARGEST JUMPS

Figures A.1 and A.2 show the time-to-solution and load-balance metric for the binary tree, regular, load-balanced, and load-balanced-by-dimension cuts on the unbalanced pin mesh and "heavier" unbalanced pin mesh, respectively. Figure A.3 shows the Level-2 mesh sweep times from the time-to-solution estimator and PDT, as well as the load-balance metric for (1) regular cuts, (2) hand-balanced cuts, (3) load-balanced cuts, (4) load-balanced-by-dimension cuts, and (5) binary tree cuts.

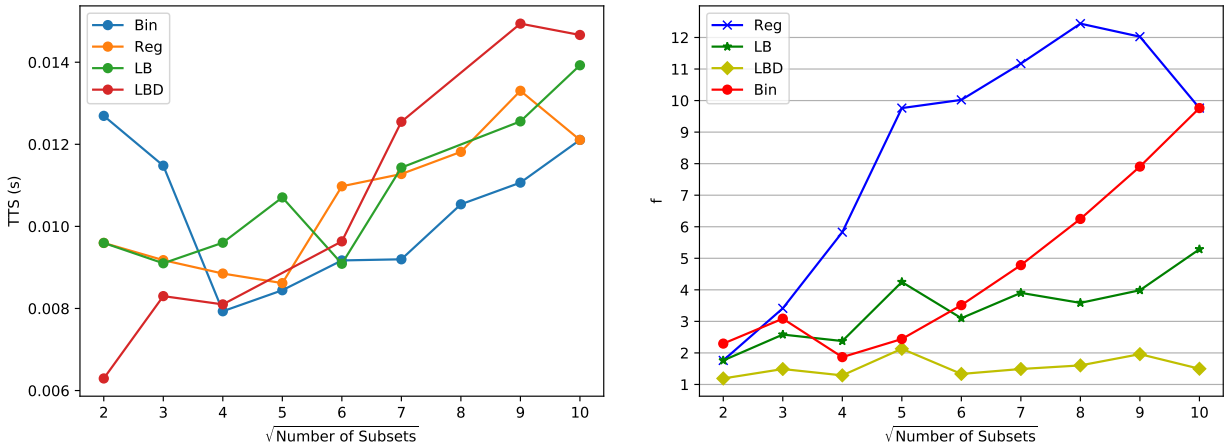


Figure A.1: The time-to-solution and load-balance metric for binary tree, regular, load-balanced and load-balanced-by-dimension cuts on the unbalanced pins mesh from 2 to 10 subsets in each dimension.

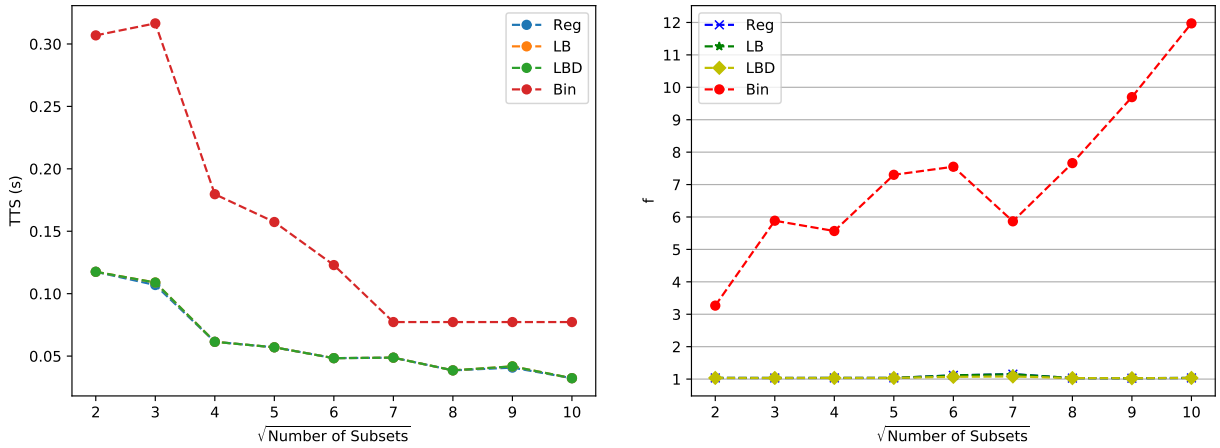
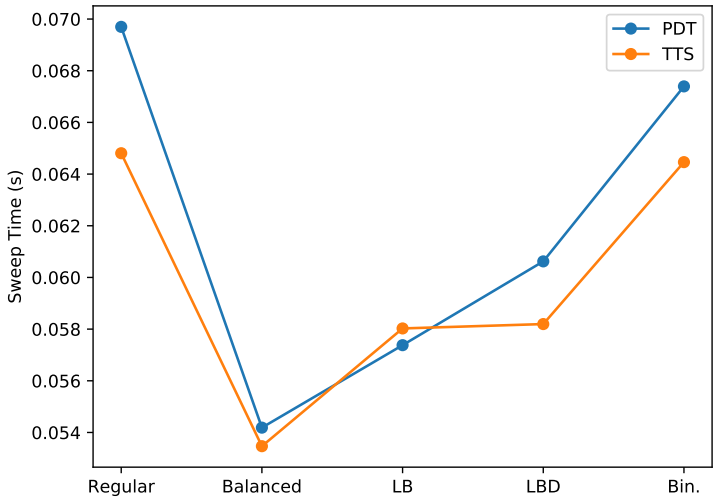


Figure A.2: The time-to-solution and load-balance metric for binary tree, regular, load-balanced and load-balanced-by-dimension cuts on the "heavier" unbalanced pins mesh from 2 to 10 subsets in each dimension.



Type	f
Regular	32.62
Balanced	2.38
LB	5.01
LBD	1.99
Bin.	21.96

Figure A.3: The Level-2 mesh sweep times from the time-to-solution estimator for (1) regular cuts, (2) hand-balanced cuts, (3) load-balanced cuts, (4) load-balanced-by-dimension cuts, and (5) optimized cuts. Additionally, the load-balance metric f is reported for each partition type.