RETURN TO ME WHEN DONE
DO NOT THROW AWAY

uh?

PARTITIONING OPTIMIZATION FOR MASSIVELY PARALLEL TRANSPORT SWEEPS ON UNSTRUCTURED GRIDS

lit review: only 7 references! I will expect ~50 in the final document (at least)

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,	John E. Hurtado

December 2019

Major Subject: Nuclear Engineering

ABSTRACT

This is the first numbered page, lower case Roman numberal (ii). Page numbers are outside the prescribed margins, at the bottom of the page and centered; everything else is inside the margins.No bold on this page (Exception: heading ABSTRACT is bold if major headings are bold. *This LaTeX template applies to this exception*).

Text begins two double spaces below the major heading. Recommended length of text is no more than 350 words. Vertical spacing is double spaced or space-and-a-half. (*This LaTeX template applies double space for this ABSTRACT.*) The same margin settings and text alignment are followed else where in this thesis. There should be no numbered references or formal citations in ABSTRACT.

The content of this ABSTRACT provides a complete, succinct snapshot of the research, addressing the purpose, methods, results, and conclusions of the research. As a result, it should stand alone without any formal citations or references to chapters/sections of the work. To accomodate with a variety of online database, images or complex equations should also be avoided.

The next pages are Dedication, Acknowledgments, Contributors and Funding Sources, and Nomenclature. Of these, Contributors and Funding Sources is required. The rest are optional.

DEDICATION

To my mother, my father, my grandfather, and my grandmother. To see what happens with multiple lines, I extend this next part into a second line.

ACKNOWLEDGMENTS

This section is also optional, limited to four pages. It must follow the Dedication Page (or Abstract, if no Dedication). If listing preliminary pages in Table of Contents, include Acknowledgments. Heading (ACKNOWLEDGMENTS) is bold if major headings are bold. It should be in same type size and style as text. So does vertical spacing, paragraph style, and margins. Also, ensure that the spelling of "acknowledgments" matches throughout the text and the table of contents.

I would like to thank the Texas A&M University Office of Graduate and Professional Studies to allow me to construct this LaTeX thesis template. Special thanks to JaeCee Crawford, Amy Motquin, Ashley Schmitt, Rachel Krolczyk, and Roberta Caton for carefully reviewing this material.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis (or) dissertation committee consisting of Professor XXXX [advisor –âĂŞ also note if co-advisor] and XXX of the Department of [Home Department] and Professor(s) XXXX of the Department of [Outside Department].

The data analyzed for Chapter X was provided by Professor XXXX. The analyses depicted in Chapter X were conducted in part by Rebecca Jones of the Department of Biostatistics and were published in (year) in an article listed in the Biographical Sketch.

All other work conducted for the thesis (or) dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by a fellowship from Texas A&M University and a dissertation research fellowship from XXX Foundation.

NOMENCLATURE

OGAPS	Office of Graduate and Professional Studies at Texas A&M University
B/CS	Bryan and College Station
TAMU	Texas A&M University
SDCC	San Diego Comic-Con
EVIL	Every Villain is Lemons
EPCC	Educator Preparation and Certification Center at Texas A&M University - San Antonio
FFT	Fast Fourier Transform
ARIMA	Autoregressive Integrated Moving Average
SSD	Solid State Drive
HDD	Hard Disk Drive
O&M	Eller Oceanography and Meteorology Building
DOS	Disk Operating System
HDMI	High Definition Multimedia Interface
L^1	Space of absolutely Lebesgue integrable functions; i.e., $\int \lvert f \rvert < \infty$
L^2	Space of square-Lebesgue-integrable functions, i.e., $\int \lvert f \rvert^2 < \infty$
$PC(S)$	Space of piecewise-continuous functions on S
GNU	GNU is Not Unix
GUI	Graphical User Interface
PID	Principal Integral Domain
MIP	Mixed Integer Program

LP	Linear Program

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

This is not the proper way of introducing the topic. The introduction should deal with the general context, little equations, explain what the problem at hand is, give its background (how was it solved before; if not, where is the problem coming from This part would include some liter. review! Finally, the introduction concludes by introducing the other chapters. In my view, the introduction is missing.

1. INTRODUCTION

The steady-state neutron transport equation describes the behavior of neutrons in a medium and is given by Eq. (1.1):

$$\vec{\Omega}\cdot\vec{\nabla}\psi(\vec{r},E,\vec{\Omega})+\Sigma_t(\vec{r},E)\psi(\vec{r},E,\vec{\Omega}) = \int_0^\infty dE' \int_{4\pi} d\Omega'\Sigma_s(\vec{r},E'\rightarrow E,\Omega'\rightarrow\Omega)\psi(\vec{r},E',\vec{\Omega}')+S_{ext}(\vec{r},E,\vec{\Omega}), \tag{1.1}$$

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. (1.1) represents the gain terms, where S_{ext} is the external source of neutrons and $\int_0^\infty dE' \int_{4\pi} d\Omega'\Sigma_s(E'\rightarrow E,\Omega'\rightarrow\Omega)\psi(\vec{r},E',\vec{\Omega}')$ 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, $\Sigma_s(E'\rightarrow E,\Omega'\rightarrow\Omega)$, has its angular dependence present in the integral, and is divided by 4π to reflect isotropic behavior. This yields:

$$\begin{aligned}\vec{\Omega}\cdot\vec{\nabla}\psi(\vec{r},E,\vec{\Omega})+\Sigma_t(\vec{r},E)\psi(\vec{r},E,\vec{\Omega}) &= \frac{1}{4\pi}\int_0^\infty dE'\Sigma_s(\vec{r},E'\rightarrow E)\int_{4\pi} d\Omega'\psi(\vec{r},E',\vec{\Omega}')+S_{ext}(\vec{r},E,\vec{\Omega})\\ &= \frac{1}{4\pi}\int_0^\infty dE'\Sigma_s(\vec{r},E'\rightarrow E)\phi(\vec{r},E')+S_{ext}(\vec{r},E,\vec{\Omega}),\end{aligned} \tag{1.2}$$

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

$$\phi(\vec{r},E') = \int_{4\pi} d\Omega'\psi(\vec{r},E',\vec{\Omega}'). \tag{1.3}$$

The next step to solving the transport equation is to discretize in energy, yielding Eq. (1.4), the

multigroup transport equation:

$$\vec{\Omega}\cdot\vec{\nabla}\psi_g(\vec{r},\vec{\Omega})+\Sigma_{t,g}(\vec{r})\psi_g(\vec{r},\vec{\Omega}) = \frac{1}{4\pi}\sum_{g'}\Sigma_{s,g'\to g}(\vec{r})\phi_{g'}(\vec{r})+S_{ext,g}(\vec{r},\vec{\Omega}), \quad \text{for } 1\leq g\leq G \tag{1.4}$$

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

Next, we discretize in angle using the discrete ordinates method[1], 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 $\vec{\Omega}_m$:

$$\vec{\Omega}_m\cdot\vec{\nabla}\psi_{g,m}(\vec{r})+\Sigma_{t,g}(\vec{r})\psi_{g,m}(\vec{r}) = \frac{1}{4\pi}\sum_{g'}\Sigma_{s,g'\to g}(\vec{r})\phi_{g'}(\vec{r})+S_{ext,g,m}(\vec{r}), \tag{1.5}$$

where the subscript m is introduced to describe the angular flux in direction m. ~~The subscript is not added to our inscatter term because of the isotropic scattering assumption and because the scalar flux does not depend on angle. However,~~ 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:

$$\phi_g(\vec{r})\approx\sum_{m=1}^{m=M} w_m\psi_{g,m}(\vec{r}). \tag{1.6}$$

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:

$$\vec{\Omega}_m\cdot\vec{\nabla}\psi_m(\vec{r})+\Sigma_t(\vec{r})\psi_m(\vec{r}) = \frac{1}{4\pi}\Sigma_s(\vec{r})\phi(\vec{r})+q_m^{ext+inscat}(\vec{r}) = q_m(\vec{r}), \tag{1.7}$$

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 by a simplified transport equation Eq. (1.8):

$$\vec{\Omega}_m\cdot\vec{\nabla}\psi_m^{(l+1)}(\vec{r})+\Sigma_t\psi_m^{(l+1)}(\vec{r}) = q_m^{(l)}(\vec{r}), \tag{1.8}$$

where the right hand side terms of Eq. (1.5) 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 scalar flux.

After the angular and energy dependence have been accounted for, Eq. (1.8) must be discretized in space as well. This is done by meshing the domain and utilizing one of three popular discretization techniques: finite difference[2], finite volume[2], or discontinuous finite element[3], 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 1.1 shows the sweep ordering for a given direction on both a structured and unstructured mesh.

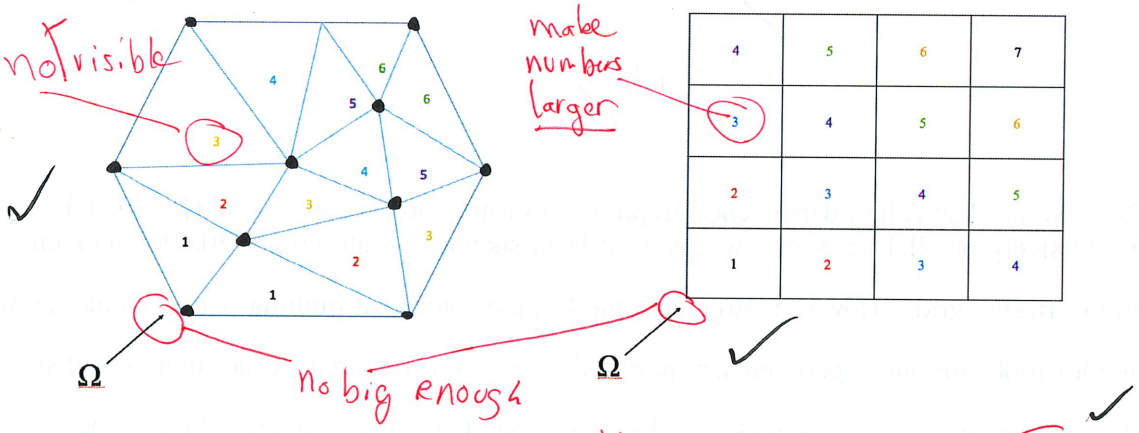


Figure 1.1: A demonstration of a sweep on a structured and unstructured meshes

The number in each cell represents the order in which the cells are solved. All cells must receive the solution downwind from them before they can begin solving. This process can be represented and stored as a task dependence graph, shown in Fig. 1.2.

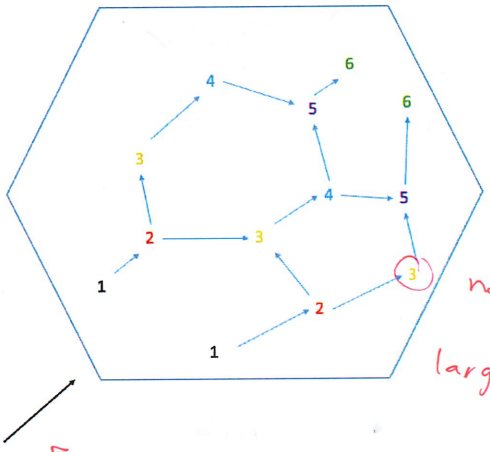


Figure 1.2: A task dependence graph of the unstructured mesh example in Fig. 1.1.

Fix

Massively parallel transport sweeps have been shown to scale up to 750,000 cores on logically Cartesian grids. However, structured meshes are somewhat limiting when simulating more complex problems and experiments, requiring the use of unstructured meshes in transport sweeps. While unstructured meshes provide the ability to simulate realistic problems, they introduce some challenges such as unbalanced partitions, which can increase the time to solution. To combat this, PDT, Texas A&M University's massively parallel deterministic transport code, introduced two load balancing algorithms that rely on moving the spatial partition boundaries, or cut planes (cut lines in 2D), throughout the mesh in order to obtain a roughly equivalent amount of cells (and therefore work) per processor. However, this sacrifices the optimal partitioning scheme[4] in favor of balance. We propose a method that weighs ideal load balancing against the consequences to the transport sweep in order to achieve the best possible time to solution.

honestly, more description of parallelism is general would be good.

2. PARALLEL TRANSPORT SWEEPS

As mentioned in the previous section, 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 task-dependence graph gives the order in which cells are solved, as shown in Fig. 1.2. The 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 cases.

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[4]:

- 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.

The basic concepts of parallel transport sweeps, partitioning, aggregation, and scheduling, are most easily described in the context of a structured transport sweep that takes place on a Cartesian mesh. Furthermore, the work proposed utilizes 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 of the domain. 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), $8N_m$ direction-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 (2.1) approximately defines parallel sweep efficiency:

$$\begin{aligned} \epsilon &\approx \frac{T_{\text{task}} N_{\text{tasks}}}{[N_{\text{stages}}][T_{\text{task}} + T_{\text{comm}}]} \\ &\approx \frac{1}{[1 + \frac{N_{\text{idle}}}{N_{\text{tasks}}}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]}, \end{aligned} \tag{2.1}$$

where N_{idle} is the number of idle stages for each 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 (2.2) and (2.3) show how T_{comm} and T_{task} are calculated:

$$T_{\text{comm}} = M_L T_{\text{latency}} + T_{\text{byte}} N_{\text{bytes}} \tag{2.2}$$

$$T_{\text{task}} = A_x A_y A_z A_m A_g T_{\text{grind}}, \tag{2.3}$$

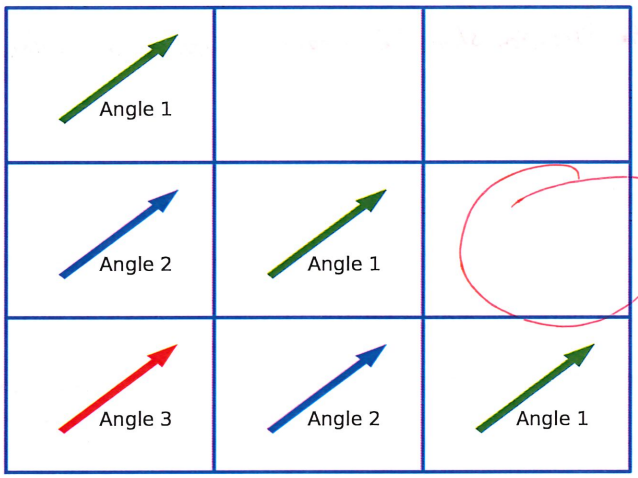
where T_{latency} is the message latency time, T_{byte} is the time required to send one byte of message, N_{bytes} is the total number of bytes of information that a processor must communicate to its downstream neighbors at each stage, and T_{grind} is the time it takes to compute a single cell, direction, 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. 2.3 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 some transport sweep algorithms is necessary. Our method will expand on PDT's sweep algorithm[4], which is an extension of the popular KBA algorithm[5].

2.1 The KBA Algorithm

Several parallel transport sweep codes use KBA partitioning in their sweeping, such as Denovo [1] and PARTISN [6]. The KBA partitioning scheme and algorithm was developed by Koch, Baker, and Alcouffe [5].

The KBA algorithm traditionally chooses $P_z = 1, A_m = 1, G = A_g = 1, A_x = N_x/P_x, A_y = N_y/P_y$, with A_z being the selectable number of z-planes to be aggregated into each task. This partitions the domain into longer, thin columns. With $N_k = N_z/A_z$, each processor performs $N_{\text{tasks}} = 8MN_k$ tasks. KBA uses a "pipelining" or assembly line approach where new work is started before old work is fully completed. Figure 2.1 shows an example of pipelining the angular work of a quadrant.



explain in caption what each square is. (cells owned by 1 proc

Figure 2.1: An example showing the pipelining of angular work from the lower left quadrant.

In Fig. 2.1, we see that as soon as a processor 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 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 simultaneously solved, and upon completion the opposing octant solves them. This is then done for the remaining octant pairs. The KBA parallel efficiency[4] for "successive in angle, successive in quadrant" is:

not defined.

$$\epsilon_{KBA} \approx \frac{1}{[1+\frac{4(P_x+P_y-2)}{8MN_k}][1+\frac{T_{\text{comm}}}{T_{\text{task}}}]}. \tag{2.4}$$

unclear. how would that happen?

2.2 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 intersecting 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, the task with $\Omega_x > 0$ wins.
3. If multiple tasks have $\Omega_x > 0$, then the task with $\Omega_y > 0$ wins.
4. 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 2.5, 2.6, and 2.7 define N_{fill}, N_{idle}, and N_{tasks}:

$$N_{\text{fill}} = \frac{P_x}{2} - 1 + \frac{P_y}{2} - 1 + N_k(\frac{P_z}{2} - 1) \tag{2.5}$$

$$N_{\text{idle}} = 2N_{\text{fill}} \tag{2.6}$$

$$N_{\text{tasks}} = 8N_m N_g N_k \tag{2.7}$$

Plugging these definitions into Eq. 2.1, the PDT optimal parallel efficiency[4] is:

$$\epsilon \approx \frac{1}{[1 + \frac{P_x+P_y-4+N_k(P_z-2)}{8MGN_k/(A_m A_G)}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]} \tag{2.8}$$

not defined at this stage. needs a little something

3. LOAD BALANCING UNSTRUCTURED MESHES

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, causing longer and unmanageable runtimes.

To combat imbalanced partitions, two load-balancing algorithms were implemented, referred to in this paper as the original load-balancing algorithm 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 for 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.
- The subsets are distributed amongst the processor domain.

A key feature is these subsets is missing. Explain what is the purpose of cutting in such a way...

Here, related subsets and cell sets (a notion that was introduced later) so that the reader can organize their thoughts.

Figure 3.1 shows an example of an unstructured mesh partitioned into four subsets in PDT.

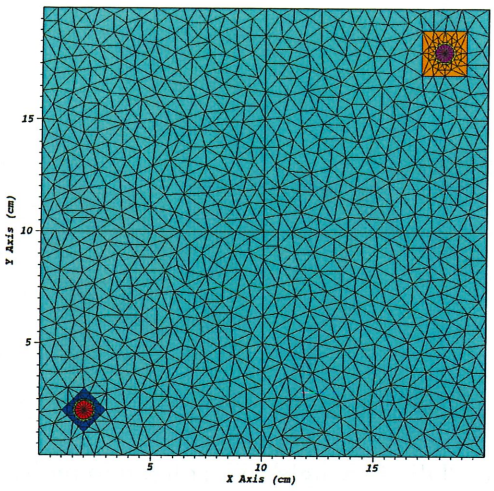


Figure 3.1: An unstructured mesh partitioned into four subsets with cut planes at $x, y = 10$ cm.

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

$$f = \frac{\max_{ijk}(N_{ijk})}{\frac{N_{tot}}{I \cdot J \cdot K}}, \tag{3.1}$$

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 direction, 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, $f = 1$.

Dimensional sub-metrics are defined to assist with the movement of the partitions:

$$f_D = \max_{d}[\sum_{d2,d3} N_{ijk}]/\frac{N_{tot}}{D}. \tag{3.2}$$

f_D is calculated by taking the maximum number of cells per row, column, or plane and dividing it by the average number of cells per the corresponding dimension. If this number is greater than predefined tolerances, the cut planes in the respective dimension are redistributed.

Figure 3.2 illustrates an example of redistributing the cut planes in x to balance the cells per

column.

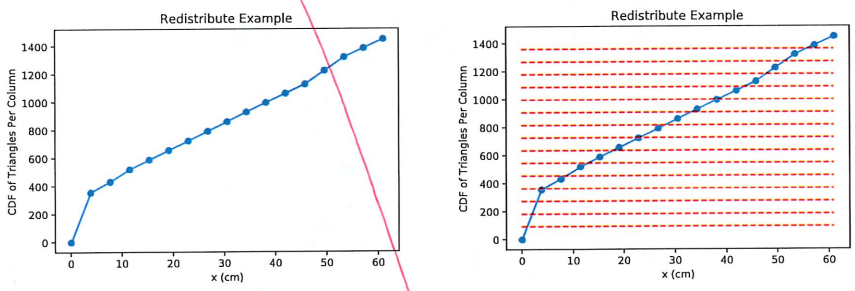


Figure 3.2: The use of the CDF of triangles per column to redistribute the cut planes in X.

The image on the left side of Fig. 3.2 shows the CDF of the triangles per column. The red lines on the right side of Fig. 3.2 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 planes are redistributed to.

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

Algorithm 1 The original load balancing algorithm.

while $f > \text{tol}_{\text{subset}}$ **do**
  if $f_I > \text{tol}_{\text{col}}$ **then**
    Redistribute the X cut planes.
  end if
  if $f_J > \text{tol}_{\text{row}}$ **then**
    Redistribute the Y cut planes.
  end if
end while

number of ← same

While the problem is not balanced, we check if the cells per column and cells per row, represented by f_I and f_J (defined by Eq. (3.2)), are balanced. If they are not, we redistribute the corresponding cut planes.

lines (you just said it was done in 2D....)

The original load-balancing algorithm placed cut planes 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 optimal sweep partitioning described by Adams et. al [?]. However, there are theoretical limits to load balancing in this fashion. Figure 3.3 shows a simple 2D subset layout with M unaligned patches of high mesh density.

don't you th this sh come mu earlier, again, th is not a progress report (here, I mean chronology is not th necessarily importa what matter is how everything ties together I think that sentence can stay, but the notion it refers to must be introd earlier

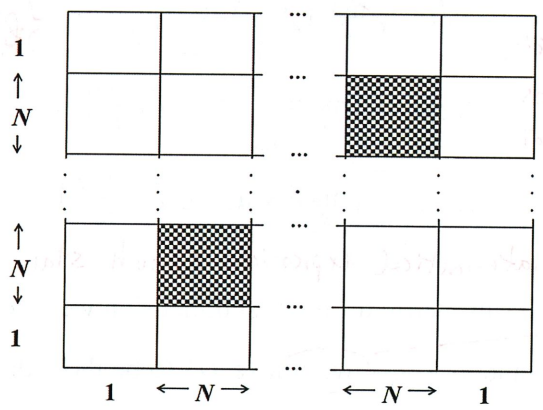


Figure 3.3: A 2D subset layout with M unaligned patches of high mesh density.

the figure does not mention M. People have no way of understanding what's going on.

undefined

The subset layout is $[M(N+1)+1] \times [M(N+1)+1]$, but only MN^2 subset have significant work, leading to a theoretical limit for the load imbalance factor:

I disagree that this is the subset layout.

not clear; express differently, using your definition of f introduced earlier. How do you expect people to follow?

$$f = \frac{(M(N+1)+1)^2}{MN^2} \xrightarrow{N\to\infty} \frac{M^2N^2}{MN^2} = M. \tag{3.3}$$

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

3.2 Load Balancing by Dimension Algorithm

The load balancing by dimension algorithm, like the original load balancing algorithm, relies on the movement of cut planes to redistribute mesh cells in a more balanced manner. However rather than moving all cut planes in all dimensions in one iteration, one dimension is balanced first for all iterations, and then the cut planes that yield the best metric for that dimension are chosen. Then the next dimensions is balanced within the first dimension. For example, if the x cut planes are balanced first, the y cut planes are balanced *within* each column. We slightly alter the definition for our dimensional sub-metrics (Eq. (3.2)) accordingly:

$$f_{D1} = \max_{d1}[\sum_{d2,d3} N_{ijk}]/\frac{N_{tot}}{D1}, \tag{3.4}$$

$$f_{D2,d1} = \left(\max_{d2}[\sum_{d3} N_{ijk,d1}]/\frac{N_{d1}}{D2}\right), \tag{3.5}$$

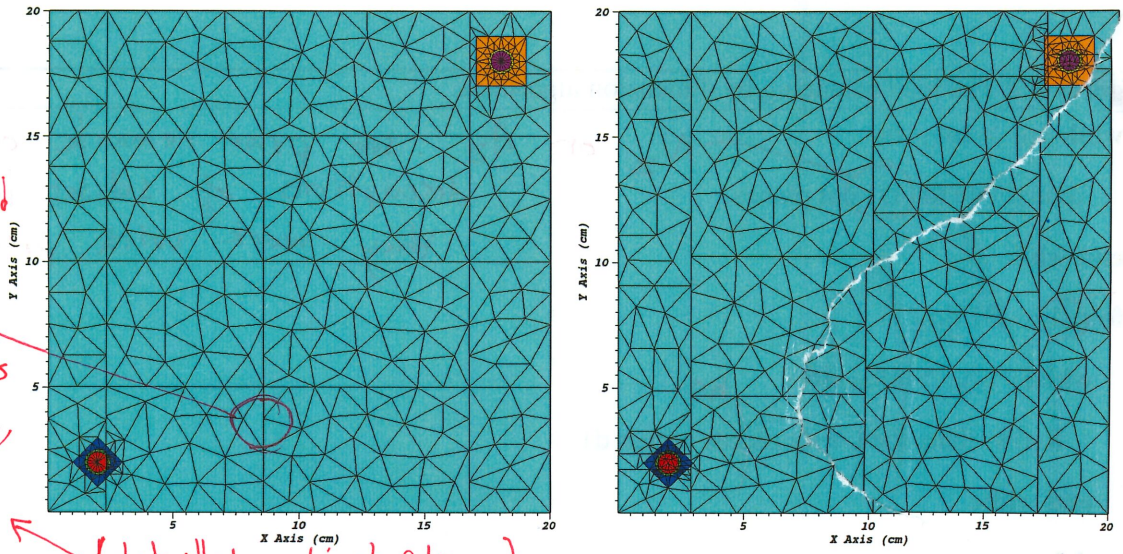
$$f_{D3,d2,d1} = \left(\max_{d3}[N_{ijk,d1,d2}]/\frac{N_{d1,d2}}{D3}\right). \tag{3.6}$$

Equation (3.4) represents the sub-metric for the first dimension we load balance. In order to assist in the explanation of $f_{D2,d1}$ and $f_{D3,d2,d1}$, we'll take $D1$ to be the z dimension. In Eq. (3.4), $D1$ represents the number of z planes, N_{tot} represents the number of cells in the mesh, and N_{ijk} is the number of cells in subset (i, j, k). f_{D1} is the maximum number of cells per plane by the average number of cells per plane.

Once the z planes are balanced according to f_{D1}, we balance the second dimension (in this example, the columns, or x cut planes) *within* each z plane. $f_{D2,d1}$ represents the column-wise sub-metrics for each z plane. In other words, there are $D1$ column-wise sub-metrics, one for each plane. These column-wise sub-metrics are calculated by dividing the maximum number of cells per column divided by the average number cells per column *in each plane*.

Once the columns within each z-plane are balanced, we balance the rows *within* the columns *within* each plane. The row-wise sub-metrics for each plane for each column, $f_{D3,d2,d1}$, are calculated by dividing the maximum number of cells per row per column by the average number of

appropriate columns in order to better balance the problem.



You never explained stitching across subsets - Yet, here you must say it is essnt. Therefore, you must have a prior discussion about stitching somewhere earlier.

(detailed, not just 2 lines)

Figure 3.4: The partitions of a problem after the original load balancing (left) and the load balancing by dimension (right) algorithms.

You cannot change tense like this. The previous paragraph was using the present tense. I think you should stick with present for your work. Amend throughout as necessary.

3.3 RESULTS

To show the behavior of the original load balancing algorithm and the load balancing by dimension algorithm, the problem shown in Fig. 3.1 was run with a varying number of subsets and with a varying maximum triangle area, with the minimum allowable angle per triangle was kept constant at 20°. We varied the number of subsets in x and y from 2 to 10 (with the total number of subsets varying from 4 to 100). The problem illustrated by Fig. 3.1 was chosen because it has two dense features in opposing corners with no features in between, lending itself to being an imbalanced mesh. Table 3.1 shows the results of this parametric study using the original load balancing algorithm vs. using the load balancing by dimension algorithm. Table 3.2 shows the percent improvements for both algorithms from no load balancing to running both load balancing algorithm, while Table 3.3 shows the improvement of the load balancing by dimension algorithm relative to the original load balancing algorithm.

cells per row per column in each plane. Algorithm 2 summarizes the load balancing by dimension process described.

Algorithm 2 The load balancing by dimension algorithm.

while $f_{D1} > \text{tol}_{D1}$ **do**
  Redistribute the D1 cut planes.
end while
for $d1$ in $D1$ **do**
  while $f_{D2,d1} > \text{tol}_{D2}$ **do**
    Redistribute the D2 cut planes within d1.
  end while
end for
for $d1$ in $D1$ **do**
  for $d2$ in $D2$ **do**
    while $f_{D3,d2,d1} > \text{tol}_{D3}$ **do**
      Redistribute the D3 cut planes within d2 within d1.
    end while
  end for
end for
Calculate f.

Figure 3.4 illustrates the behavior of both algorithms. In the left image of the figure, we see the partitions cutting across the entire domain, with the left-most x cut line moved into the denser geometric feature in the bottom left corner to more evenly distribute cells. In the right image of the figure, 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

Table 3.1: **The results of the parametric study using the original load balancing algorithm (left) and the load balancing by dimension algorithm (right).**

Area, $N^{1/2}$	2	3	4	5	6	7	8	9	10
Coarse	1.993	2.735	4.360	4.812	5.545	6.321	3.114	2.697	1.893
1.8	1.408	2.277	2.886	3.269	4.716	4.721	5.890	4.618	1.863
1.6	1.375	2.206	2.649	3.247	4.356	4.876	4.678	5.062	1.329
1.4	1.337	2.110	2.982	3.031	4.615	4.310	8.911	4.652	2.675
1.2	1.344	2.008	2.017	3.392	3.916	4.969	9.576	4.543	4.728
1.0	1.264	1.806	2.405	2.976	3.657	4.317	6.242	4.831	4.941
0.8	1.212	1.640	2.300	2.436	2.941	4.395	7.420	4.466	3.947
0.6	1.153	1.567	2.045	2.368	3.199	2.999	7.206	4.101	3.592
0.4	1.108	1.411	1.633	2.117	2.383	2.646	6.970	3.086	2.511
0.2	1.052	1.197	1.258	1.523	1.789	1.857	3.380	2.193	1.883
0.1	1.029	1.092	1.149	1.207	1.276	1.420	2.015	1.565	1.247
0.08	1.009	1.043	1.086	1.101	1.179	1.267	2.118	1.551	1.271
0.06	1.009	1.024	1.059	1.094	1.138	1.154	1.825	1.432	1.138
0.05	1.008	1.023	1.025	1.028	1.073	1.149	1.666	1.380	1.110
0.04	1.005	1.016	1.017	1.021	1.038	1.051	1.520	1.311	1.080
0.03	1.005	1.008	1.018	1.039	1.059	1.073	1.450	1.179	1.001
0.02	1.005	1.008	1.010	1.013	1.021	1.035	1.623	1.137	1.016
0.01	1.003	1.009	1.009	1.011	1.016	1.013	1.281	1.058	1.015

Area, $N^{1/2}$	2	3	4	5	6	7	8	9	10
Coarse	1.645	1.455	1.878	2.348	3.046	3.022	1.752	2.304	1.451
1.8	1.034	1.460	2.127	1.744	2.098	2.588	2.623	2.776	2.872
1.6	1.015	1.396	1.899	1.877	2.090	2.857	2.608	3.582	2.604
1.4	1.011	1.418	1.631	1.964	1.820	2.968	2.055	2.201	1.523
1.2	1.019	1.344	1.483	1.983	2.122	3.023	2.356	4.765	2.371
1.0	1.007	1.338	1.641	2.313	3.097	2.098	2.563	2.808	2.637
0.8	1.016	1.157	1.457	1.982	1.881	2.340	2.283	3.513	3.947
0.6	1.012	1.111	1.199	1.598	1.901	1.791	2.330	3.005	3.719
0.4	1.005	1.024	1.204	1.288	1.665	1.492	1.660	2.528	2.511
0.2	1.007	1.021	1.025	1.116	1.175	1.358	1.478	1.624	1.837
0.1	1.003	1.019	1.024	1.019	1.092	1.122	1.161	1.087	1.247
0.08	1.007	1.010	1.022	1.035	1.035	1.077	1.176	1.135	1.219
0.06	1.004	1.009	1.021	1.032	1.031	1.070	1.102	1.080	1.072
0.05	1.002	1.005	1.019	1.023	1.038	1.071	1.096	1.094	1.101
0.04	1.002	1.008	1.008	1.021	1.027	1.028	1.063	1.091	1.080
0.03	1.003	1.008	1.013	1.014	1.030	1.044	1.068	1.074	1.001
0.02	1.002	1.006	1.009	1.013	1.020	1.030	1.038	1.058	1.016
0.01	1.001	1.006	1.007	1.011	1.015	1.013	1.030	1.029	1.015

Table 3.2: **The ~~percent improvement~~ of the original load balancing algorithm (left) and the load balancing by dimension algorithm (right).**

Area, $N^{1/2}$	2	3	4	5	6	7	8	9	10
Coarse	0.000	0.367	0.403	0.552	0.628	0.491	0.890	0.720	0.765
1.8	0.000	0.091	0.337	0.364	0.473	0.390	0.767	0.413	0.683
1.6	0.000	0.093	0.398	0.368	0.499	0.370	0.815	0.353	0.774
1.4	0.000	0.061	0.080	0.410	0.415	0.412	0.570	0.413	0.545
1.2	0.000	0.007	0.391	0.340	0.378	0.315	0.536	0.245	0.196
1.0	0.000	0.038	0.206	0.420	0.341	0.186	0.696	0.201	0.160
0.8	0.000	0.049	0.109	0.336	0.434	0.139	0.637	0.228	0.000
0.6	0.000	0.000	0.057	0.199	0.163	0.346	0.517	0.000	0.090
0.4	0.000	0.000	0.065	0.013	0.267	0.147	0.528	0.179	0.000
0.2	0.000	0.000	0.000	0.000	0.001	0.041	0.566	0.121	0.000
0.1	0.000	0.000	0.000	0.000	0.000	0.000	0.540	0.089	0.000
0.08	0.000	0.000	0.000	0.000	0.000	0.000	0.458	0.000	0.000
0.06	0.000	0.000	0.000	0.000	0.000	0.000	0.409	0.000	0.000
0.05	0.000	0.000	0.000	0.000	0.000	0.000	0.360	0.000	0.000
0.04	0.000	0.000	0.000	0.000	0.000	0.000	0.348	0.000	0.000
0.03	0.000	0.000	0.000	0.000	0.000	0.000	0.293	0.000	0.000
0.02	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
0.01	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Area, $N^{1/2}$	2	3	4	5	6	7	8	9	10
Coarse	0.175	0.663	0.743	0.781	0.796	0.757	0.938	0.760	0.820
1.8	0.266	0.417	0.511	0.661	0.766	0.665	0.896	0.647	0.512
1.6	0.262	0.426	0.568	0.635	0.760	0.631	0.897	0.542	0.557
1.4	0.244	0.369	0.497	0.618	0.769	0.595	0.901	0.722	0.741
1.2	0.242	0.336	0.552	0.614	0.663	0.583	0.886	0.208	0.597
1.0	0.203	0.287	0.458	0.549	0.442	0.605	0.875	0.536	0.552
0.8	0.162	0.330	0.435	0.460	0.638	0.542	0.888	0.393	0.000
0.6	0.122	0.291	0.447	0.460	0.503	0.610	0.844	0.267	0.058
0.4	0.093	0.274	0.310	0.400	0.488	0.519	0.888	0.328	0.000
0.2	0.042	0.147	0.185	0.267	0.344	0.299	0.810	0.349	0.025
0.1	0.026	0.067	0.109	0.156	0.144	0.210	0.735	0.367	0.000
0.08	0.002	0.032	0.059	0.060	0.122	0.150	0.699	0.268	0.041
0.06	0.005	0.014	0.036	0.057	0.094	0.073	0.643	0.246	0.058
0.05	0.006	0.017	0.006	0.005	0.033	0.068	0.579	0.208	0.008
0.04	0.002	0.008	0.009	0.000	0.011	0.022	0.544	0.168	0.000
0.03	0.002	0.000	0.005	0.024	0.028	0.027	0.479	0.089	0.000
0.02	0.003	0.002	0.001	0.000	0.001	0.004	0.361	0.070	0.000
0.01	0.002	0.003	0.002	0.000	0.001	0.000	0.196	0.027	0.000

The data in Tables 3.1 and 3.2 showcase an important point. As mesh refinement increases, the need for load balancing decreases. If sparse regions of the domain have a similar number of cells to the dense regions of the domain, the problem will be inherently balanced with even partitions. Table 3.3 demonstrates that with the exception of a few outliers, the load balancing by dimension algorithm is an improvement over the original load balancing algorithms, particularly for coarser mesh refinements. The metric improves by a max of 76.9% and a mean of 21.7% with the load balancing by dimensions algorithm over the original load balancing algorithm.

correct??

Table 3.3: **The percent improvement of the load balancing by dimension algorithm over the original load balancing algorithm.**

Area, $N^{1/2}$	2	3	4	5	6	7	8	9	10
Coarse	0.175	0.468	0.569	0.512	0.451	0.522	0.437	0.146	0.234
1.8	0.266	0.359	0.263	0.466	0.555	0.452	0.555	0.399	-0.542
1.6	0.262	0.367	0.283	0.422	0.520	0.414	0.443	0.291	-0.959
1.4	0.244	0.328	0.453	0.352	0.606	0.311	0.769	0.527	0.431
1.2	0.242	0.331	0.265	0.415	0.458	0.392	0.754	-0.049	0.499
1.0	0.203	0.259	0.318	0.223	0.153	0.514	0.589	0.419	0.466
0.8	0.162	0.295	0.366	0.186	0.360	0.467	0.692	0.213	-0.000
0.6	0.122	0.291	0.414	0.325	0.406	0.403	0.677	0.267	-0.035
0.4	0.093	0.274	0.262	0.392	0.301	0.436	0.762	0.181	0.000
0.2	0.042	0.147	0.185	0.267	0.343	0.269	0.563	0.260	0.025
0.1	0.026	0.067	0.109	0.156	0.144	0.210	0.424	0.305	-0.000
0.08	0.002	0.032	0.059	0.060	0.122	0.150	0.445	0.268	0.041
0.06	0.005	0.014	0.036	0.057	0.094	0.073	0.396	0.246	0.058
0.05	0.006	0.017	0.006	0.005	0.033	0.068	0.342	0.208	0.008
0.04	0.002	0.008	0.009	0.000	0.011	0.022	0.301	0.168	-0.000
0.03	0.002	-0.000	0.005	0.024	0.028	0.027	0.263	0.089	0.000
0.02	0.003	0.002	0.001	0.000	0.001	0.004	0.361	0.070	0.000
0.01	0.002	0.003	0.002	-0.000	0.001	-0.000	0.196	0.027	-0.000

← explain. I would like to see the 3 meshes related to this.

* In this illustration, you have omitted any iteration/convergence discussion. That is not right. You must also include that. (a discussion of results, not just 1-2 lines)

* Finally, I note that there will be other test cases later (Level-2, spydergrid, IMC) I think these should all be provided here as well, with a discussion of LB versus LBD. You will use these later in T2S. Thus they need to be present here as well.

4. TIME-TO-SOLUTION ESTIMATOR

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 because of its powerful graph library, networkx. This library gives us a wide variety of graph mathematics that is optimized and easy to use.

4.1 Method

The time-to-solution estimator determines the time to sweep across a domain by:

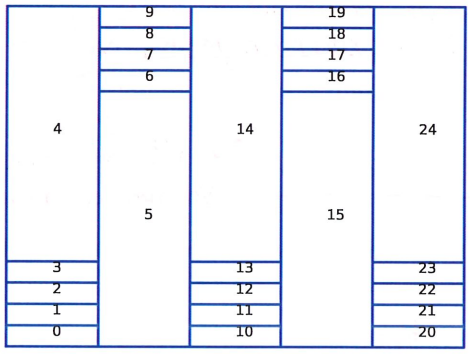
1. Given a partitioning scheme, build an adjacency matrix.
2. Build Directed Acyclic Graphs (DAGs) from the adjacency matrix, one for each quadrant/octant.
3. Weight the edges of each graph based on the solve and communication time of each subset to its neighbors.
4. Add and adjust graphs based on how many angles are pipelined.
5. Adjust the weights of each graph to operate on the universal timescale.
6. Adjust the weights of each graph to reflect sweep conflicts between octants.
7. Calculate the time to solution.

4.1.1 Building the adjacency matrices

Before building the graph for each quadrant/octant, an adjacency matrix must be built for the given partitioning scheme. The adjacency matrix proves 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 will be interchangeable, but for now, this ordering must be preserved.

Figure 4.1 shows a partitioning scheme for 5 subsets each in the x and y dimensions with its corresponding adjacency matrix.



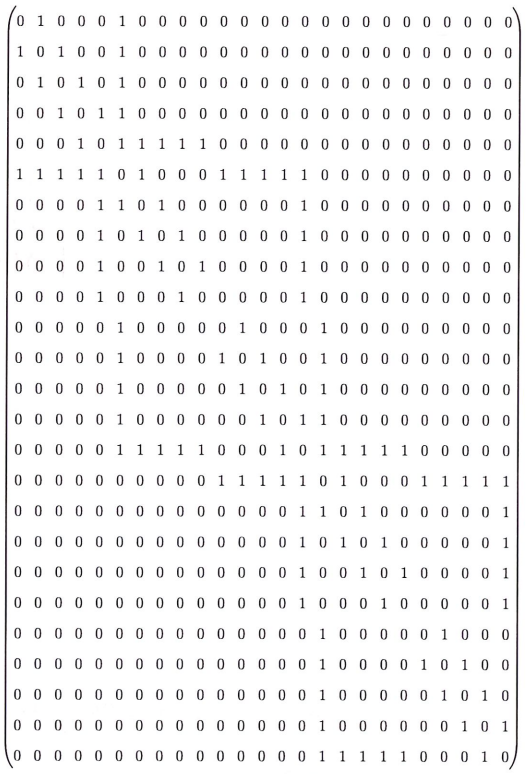


Figure 4.1: A 5x5 subset partitioning scheme and its corresponding adjacency matrix.

4.1.2 Building the directed acyclic graphs (DAGs)

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

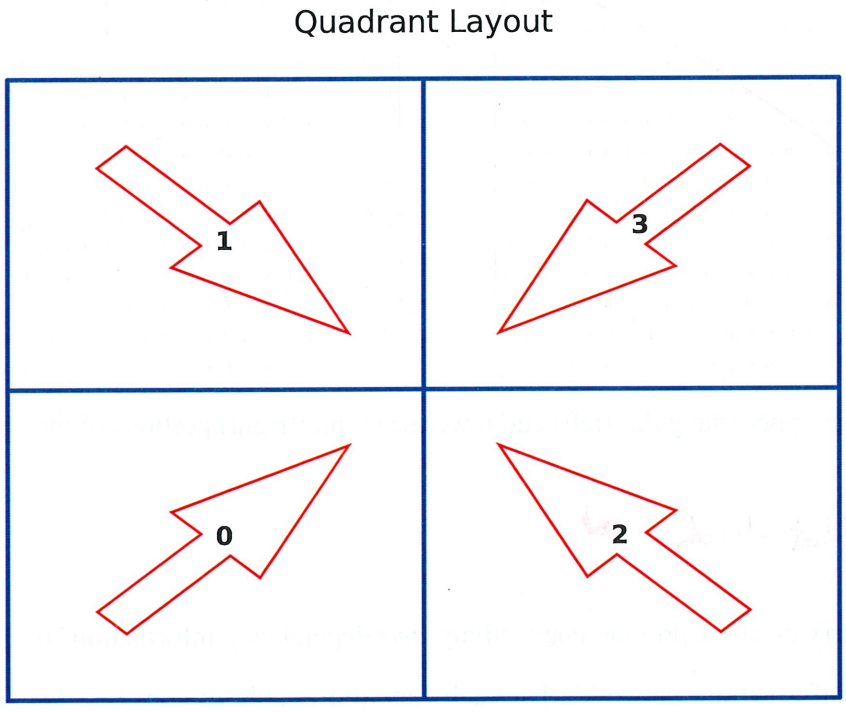
2D or 2d . be consistent

4.1.2.1 *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 4.2 illustrates this numbering.



← is that really necessary ?

Figure 4.2: 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. 4.1, pull the upper triangular and lower triangular portions of the matrix, shown in Fig. 4.3.

Figure 4.3: The upper triangular (left) and lower triangular(right) portions of the adjacency matrix in Fig. 4.1.

These matrix portions provide connectivity *and* dependency information for quadrants 0 and 3. With this information, network's DiGraph function is able to construct the DAGs for these quadrants. Figure 4.4 shows the the DAGS associated with quadrants 0 and 3, built from the adjacency matrices in Fig. 4.3.

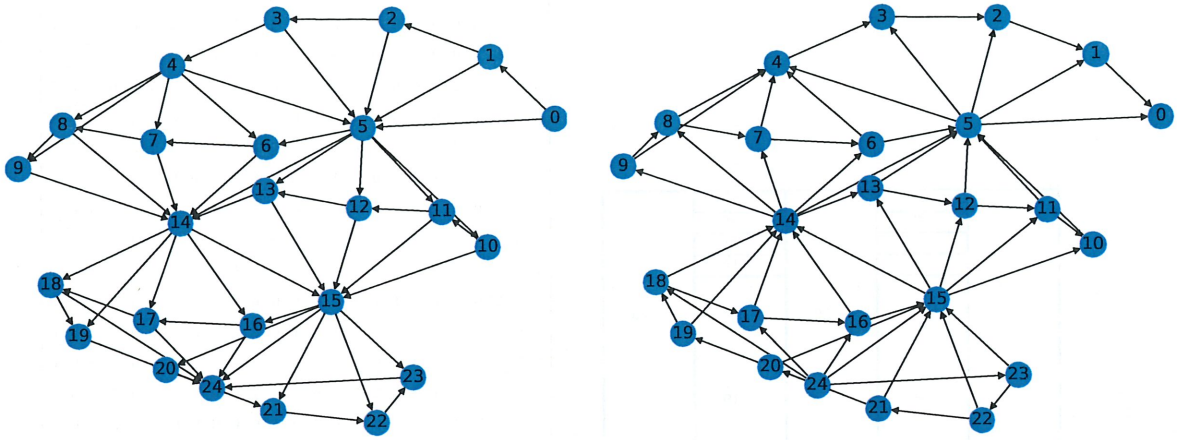


Figure 4.4: The quadrant 0 DAG (left) and the quadrant 3 DAG(right).

thankfully! is that adjective useful?

Upon inspection of these two DAGs, we see that they show the correct connectivity, dependency, and expected opposing sweep ordering. Quadrant 0 starts its sweep from subset 0, finishing at subset 24, and quadrant 3 starts its sweep from subset 24, 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. 4.5.

why temporarily? I don't get it

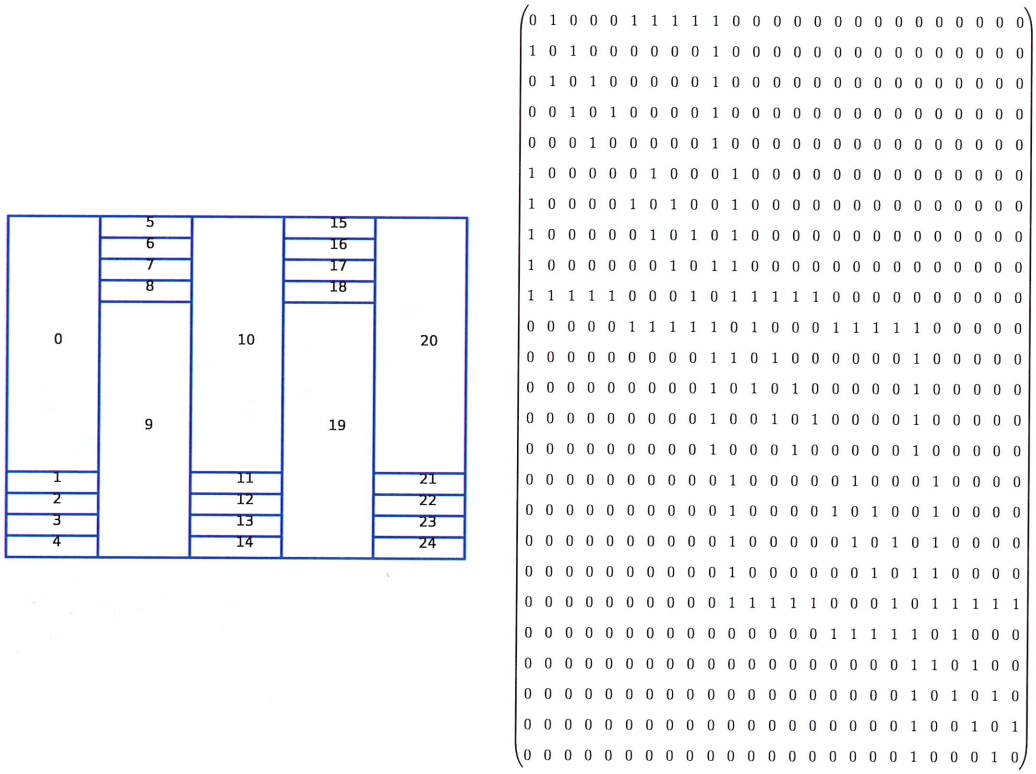


Figure 4.5: The flipped subset ordering and corresponding flipped adjacency matrix for the partitioning scheme in Fig. 4.1.

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. 4.1). Figure 4.6 shows the DAGs for quadrants 1 and 2.

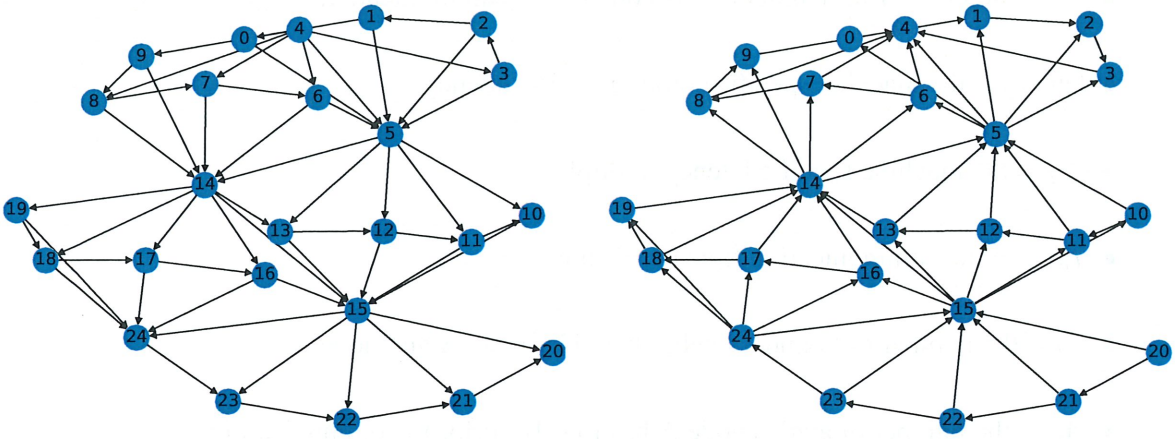


Figure 4.6: The quadrant 1 DAG (left) and the quadrant 2 DAG(right).

Upon inspection of these two DAGs, we see that they show the correct connectivity, dependency, and expected opposing sweep ordering. Quadrant 1 starts its sweep from subset 4, finishing at subset 20, and quadrant 3 starts its sweep from subset 20, finishing at subset 4.

4.1.3 Weighting the task dependence graphs

Each graph is weighted to reflect the solve 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 to node B. Equation 4.1 shows how the weight is calculated:

$$\text{weight} = \text{mcff} \cdot [T_{wu} + N_n \cdot \text{latency} \cdot M_L + T_{\text{comm}} \cdot N_b \cdot A_m \cdot upbc + \cdot N_c \cdot (T_c + A_m \cdot (T_m + T_g))], \quad (4.1)$$

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.

which you must describe in earlier chapters in more details

The weighting function is based on PDT's performance model[**?**], 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. 4.1, 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 neighboring subsets. Given a mesh density, the number of cells per subset is given by Eq 4.2:

$$\text{cells per subset} = \int_{x_i}^{x_{i+1}} \int_{y_j}^{y_{j+1}} \int_{z_k}^{z_{k+1}} \text{mesh density } dxdydz, \tag{4.2}$$

where the integral bounds represent the cut plane coordinates that form the subset. To estimate the boundary cells to each neighbor, we assume that the mesh within each subset is mostly uniform.

Equations 4.3-4.5 calculate the boundary cells along each face in 3D:

$$n_{xy} = \left(\frac{N}{V}\right)^{2/3} \cdot L_x \cdot L_y, \tag{4.3}$$

$$n_{xz} = \left(\frac{N}{V}\right)^{2/3} \cdot L_x \cdot L_z, \tag{4.4}$$

$$n_{yz} = \left(\frac{N}{V}\right)^{2/3} \cdot L_y \cdot L_z, \tag{4.5}$$

where N 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 4.6 and 4.7 show the 2-dimensional equivalents to 4.3-4.5:

$$n_x = \left(\frac{N}{A}\right) \cdot L_x, \tag{4.6}$$

$$n_y = \left(\frac{N}{A}\right) \cdot L_y, \tag{4.7}$$

where A is the subset area. With this information, we can successfully compute all weights in each graph according to Eq. 4.1. Once graphs are weighted, we add and adjust graphs for the number of angles pipelined per octant/quadrant.

4.1.4 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 8 ~~graphs per angleset~~, or 1 graph per octant per angleset.

4.1.5 Adjusting 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. For each node in each graph, the following is done:

1. Get the longest path to the node.
2. Sum this 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 it is ready to solve. This universal edge weighting is crucial for detecting and resolving conflicts during the sweep. Figure 4.7 shows a simple example of a TDG before and after this weight adjustment.

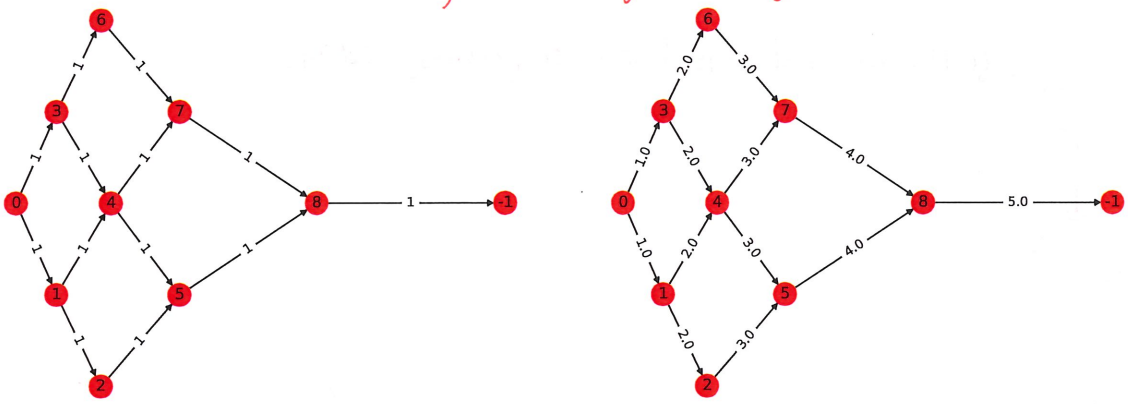


Figure 4.7: A TDG before (left) and after (right) universal edge weighting is applied.

4.1.6 Adjusting the weights of each graph to detect and resolve conflicts

At this point in the time-to-solution estimation process, we have a graph per octant/quadrant per angleset, each weighted on a universal time scale. The time to solution is best summarized as a "marching" process:

1. Starting at time $t = 0$, we find the first interaction across all graphs.
2. If at that time, multiple graphs are solving the same node, we have a conflict.
3. The graph that "wins" the conflict does not have its weights modified, while the graph that lose the conflict modify their downstream weights according to how long they are delayed.
4. Update t to the next interaction across all graphs.
5. Repeat steps 3 and 4 until all graphs are finished sweeping.

When a conflict is detected, the time-to-solution estimator defaults to 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 grater remaining depth-of-graph (simply, more work remaining), wins. In the case of a tie in the depth-of-graph, 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.

4.1.7 Estimating the final time-to-solution

Once all graphs have had their weights adjusted 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 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.

4.2 2D Verification

A verification study in 2D was run to verify the time-to-solution estimator for 2D partitioning schemes with perfectly balanced partitions. The test problems were verified against a code written by Jean Ragusa that mimics PDT's scheduler in two dimensions. For consistency, the time-to-solution estimator utilized an unweighted depth-of-graph algorithm during the verification study to match PDT's scheduling. The verification study consists of the following problems:

1. 2x2 to 10x10 subsets in x and y with regular partitions and 1 to 6 angles per quadrant.

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

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

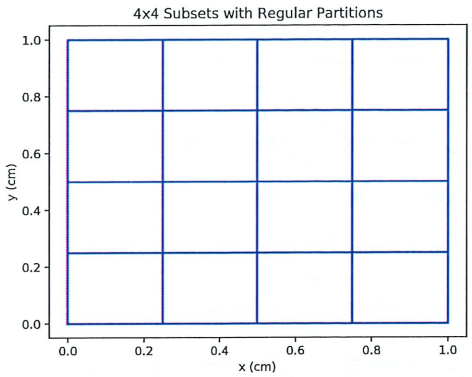
4. 2x2 to 10x10 subsets in x and y with probable worst-case partitions and 1 to 6 angles per quadrant.

"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 4.10 shows examples of this partitioning style. "Random" partitions possesses no such limitations on either set of cut lines, as shown by Fig. 4.12.

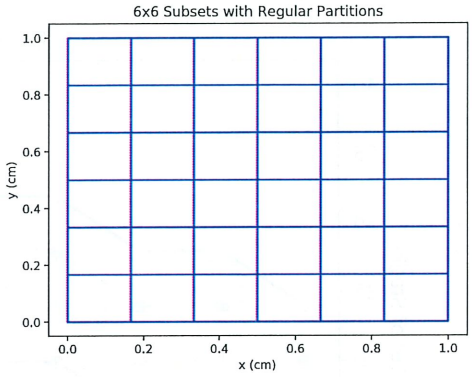
Figures 4.8, 4.10, 4.12, 4.14 show the four partitioning schemes and Figs. 4.9, 4.11, 4.13, 4.15 show the results of the verification study for each partitioning scheme. In the results, a stage is defined as the time it takes to solve all cells in a subset for an angle.

4.2.1 Regular Partitions

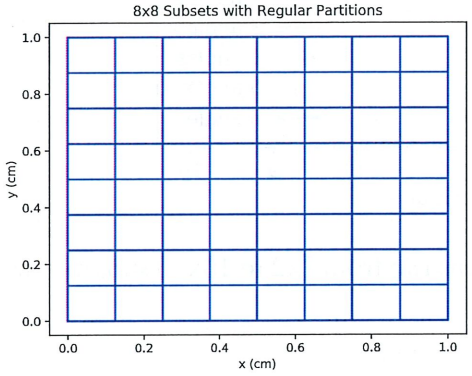
Figure 4.8 shows four examples of the regular partitioning scheme used for the first part of the verification study. Cut lines in both dimensions go all the way across the domain. This reflects the partitioning scheme after the original load balancing algorithm described in Section 3.1 is used.



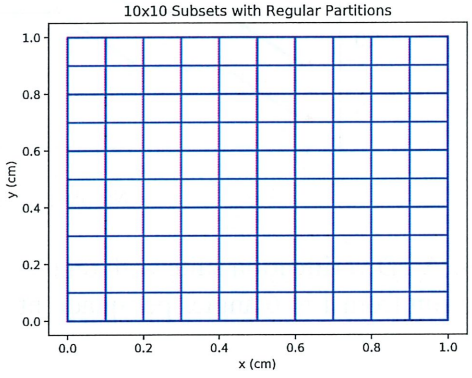
(a) 4x4 subsets with regular partitions.



(b) 6x6 subsets with regular partitions.



(c) 8x8 subsets with regular partitions.



(d) 10x10 subsets with regular partitions.

Figure 4.8: Examples of regular partitioning.

Using regular partitions as shown in Fig. 4.8, 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 4.9 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 ~~verifies perfectly~~ is in perfect agreement on regular partitions with multiple angles per quadrant.

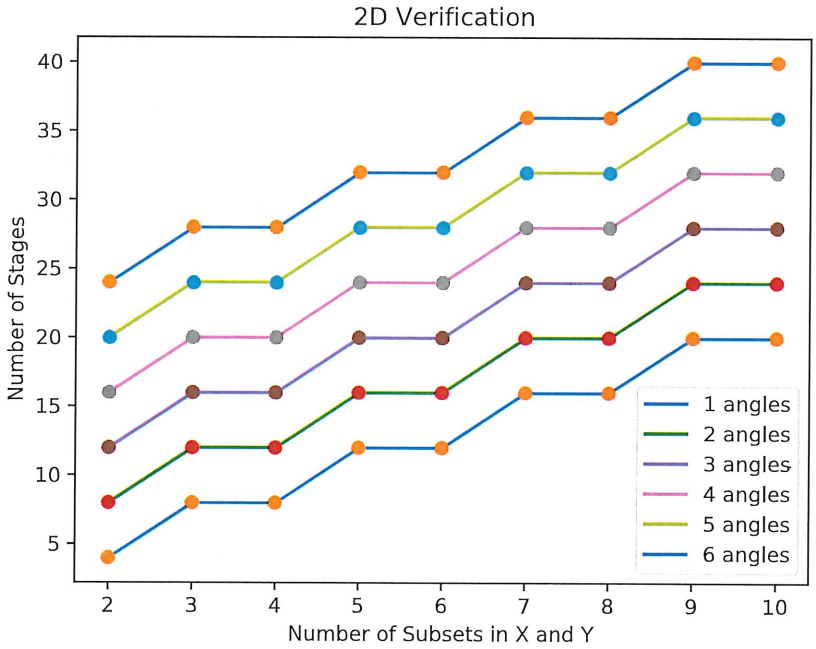
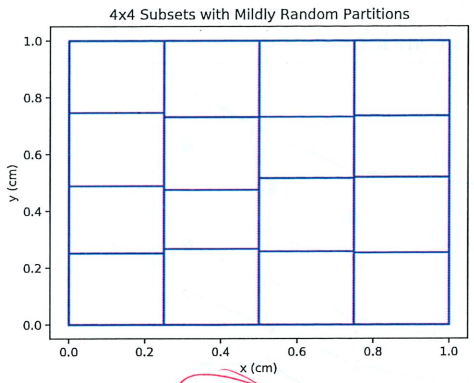


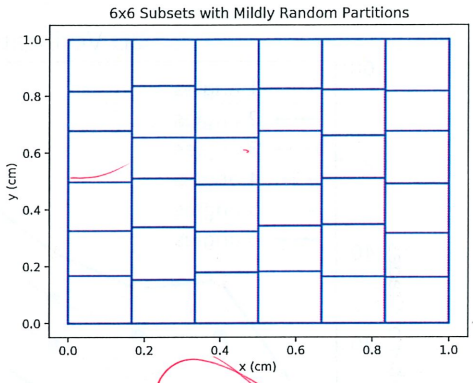
Figure 4.9: A 2D verification suite with regular partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 angles per quadrant.

4.2.2 Mildly Random Partitions

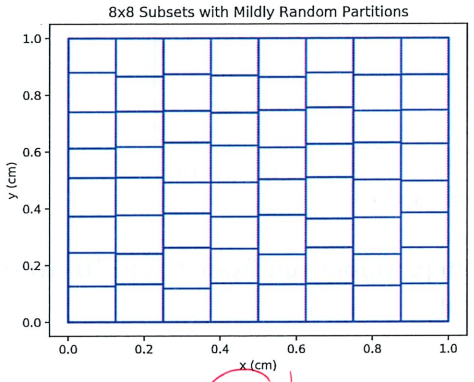
Figure 4.10 shows four examples of the mildly random partitioning scheme used for the second part of the verification study. Cut lines in the x dimension go all the way across the domain, and are uniformly distributed. This reflects a possible partitioning scheme after the load balancing by dimension algorithm described in Section 3.2 is used.



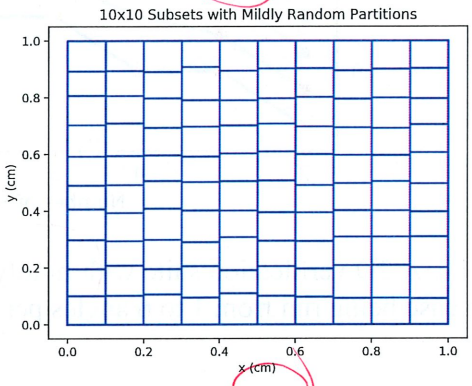
(a) 4x4 subsets with mildly random partitions.



(b) 6x6 subsets with mildly random partitions.



(c) 8x8 subsets with mildly random partitions.



(d) 10x10 subsets with mildly random partitions.

Figure 4.10: Examples of mildly random partitioning.

Using mildly random partitions as shown in Fig. 4.10, 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 4.11 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 verifies perfectly on mildly random partitions with multiple angles per quadrant.

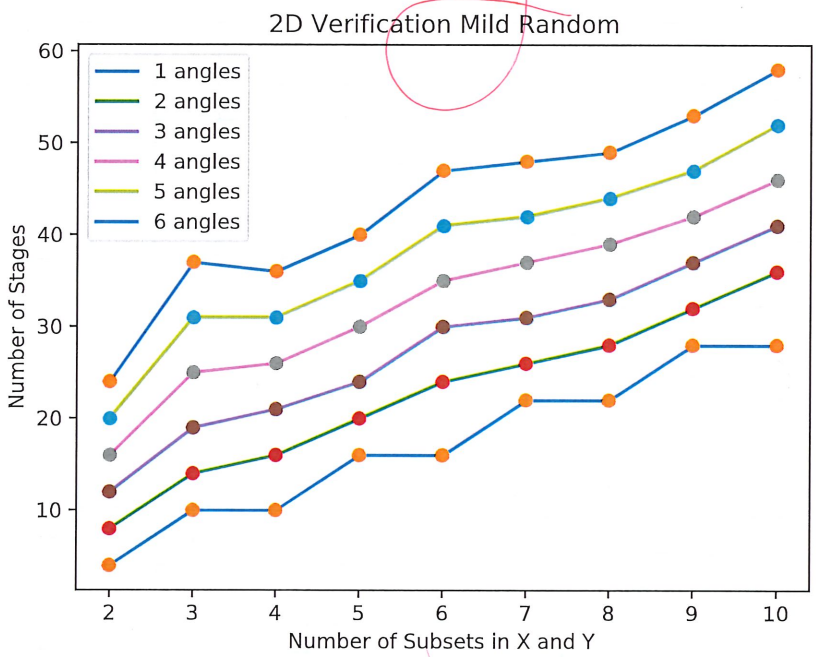
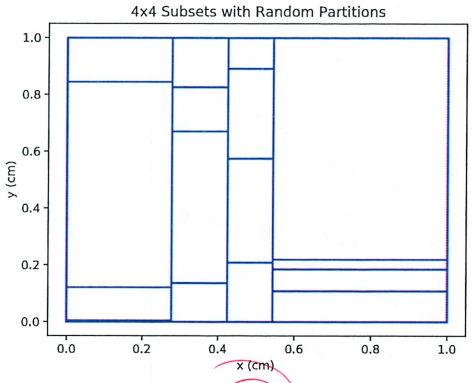


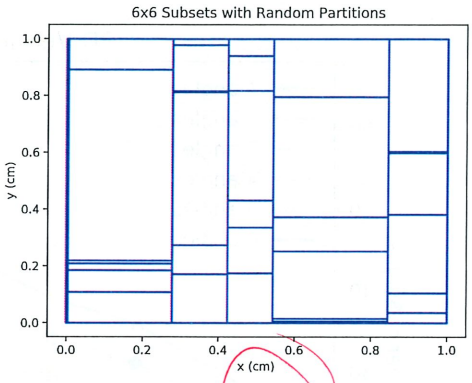
Figure 4.11: A 2D verification suite with mildly random partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 angles per quadrant.

4.2.3 Random Partitions

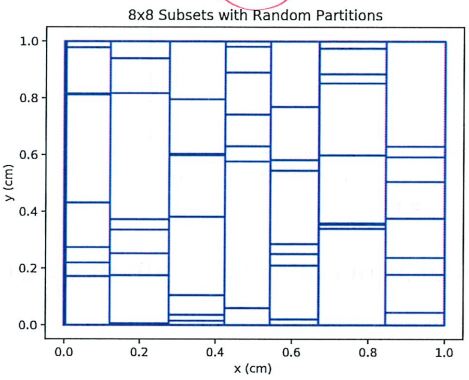
Figure 4.12 shows four examples of the random partitioning scheme used for the third part of the verification study. Cut lines in the x dimension go all the way across the domain, but are not necessarily uniformly distributed. The cut lines in y are randomly distributed in each column.This reflects a possible partitioning scheme after the load balancing by dimension algorithm described in Section 3.2 is used.



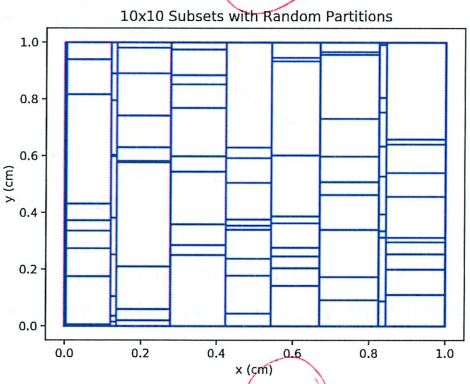
(a) 4x4 subsets with random partitions.



(b) 6x6 subsets with random partitions.



(c) 8x8 subsets with random partitions.



(d) 10x10 subsets with random partitions.

Figure 4.12: Examples of random partitioning.

Using random partitions as shown in Fig. 4.12, 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 4.13 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 verifies perfectly on random partitions with multiple angles per quadrant.

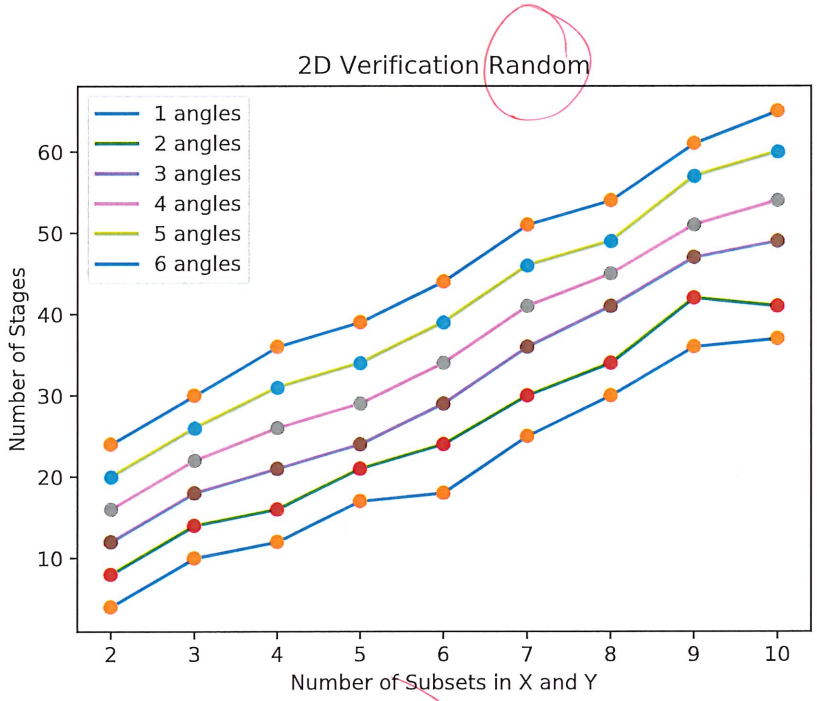
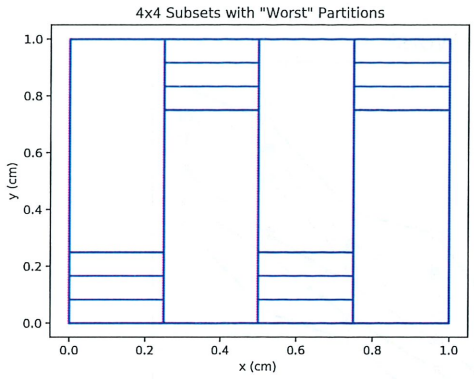


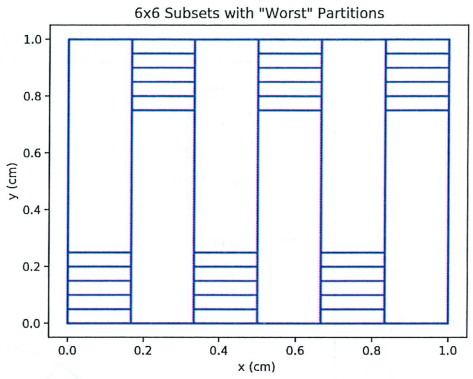
Figure 4.13: A 2D verification suite with random partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 angles per quadrant.

4.2.4 Probable Worst-Case Partitions

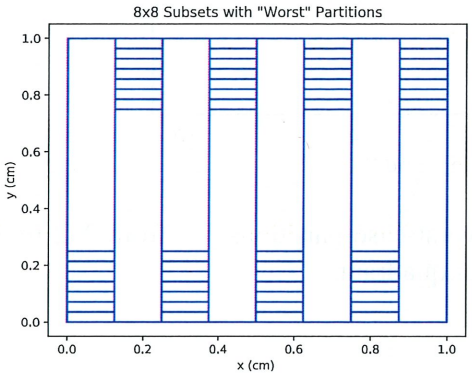
Figure 4.14 shows four examples of the probable worst-case partitioning scheme used for the final part of the verification study. Cut lines in the x dimension go all the way across the domain, and are uniformly distributed. The cut lines in y are distributed on opposing ends of alternating columns.This reflects a possible partitioning scheme after the load balancing by dimension algorithm described in Section 3.2 is used.



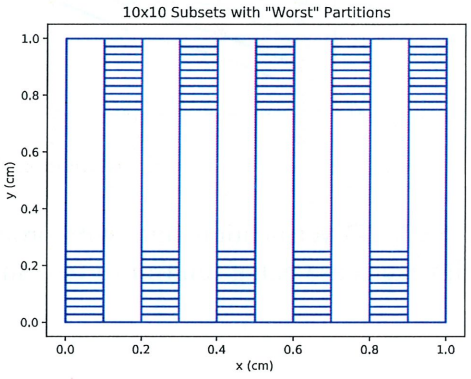
(a) 4x4 subsets with probable worst-case partitions.



(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 4.14: Examples of probable worst-case partitioning.

Using probable worst-case partitions as shown in Fig. 4.12, 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 4.15 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 verifies perfectly on probable worst-case partitions with multiple angles per quadrant.

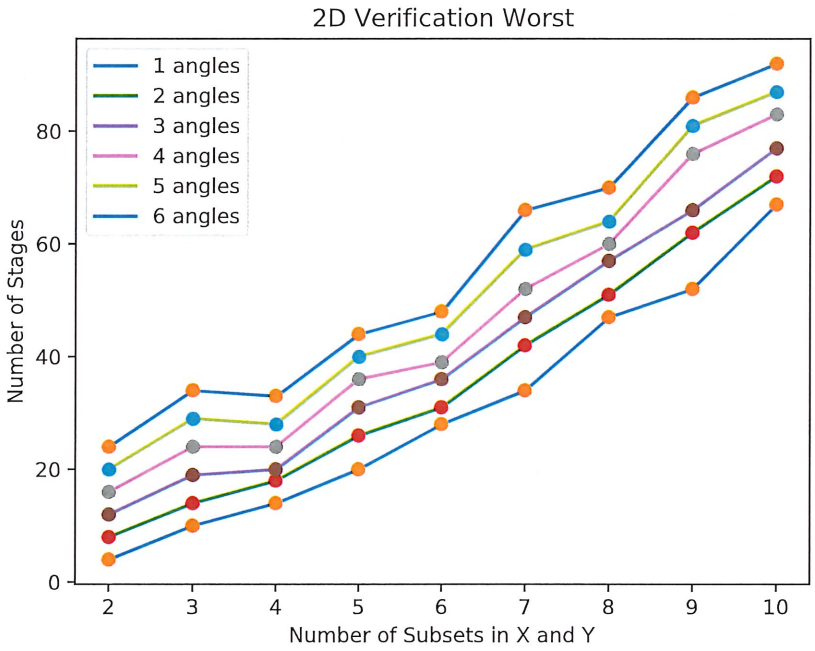


Figure 4.15: A 2D verification suite with probable worst-case partitions run from 2x2 to 10x10 subsets with each case being run from 1 to 6 angles per quadrant.

I stopped here

4.3 3D Verification

PDT's performance model was used to verify stage counts for 3D problems with regular grids. Figure 4.16 shows PDT's performance model stage counts matching perfectly with the time-to-solution estimator's stage counts for 2^3 to 10^3 subsets and from 1 to 6 angles per octant.

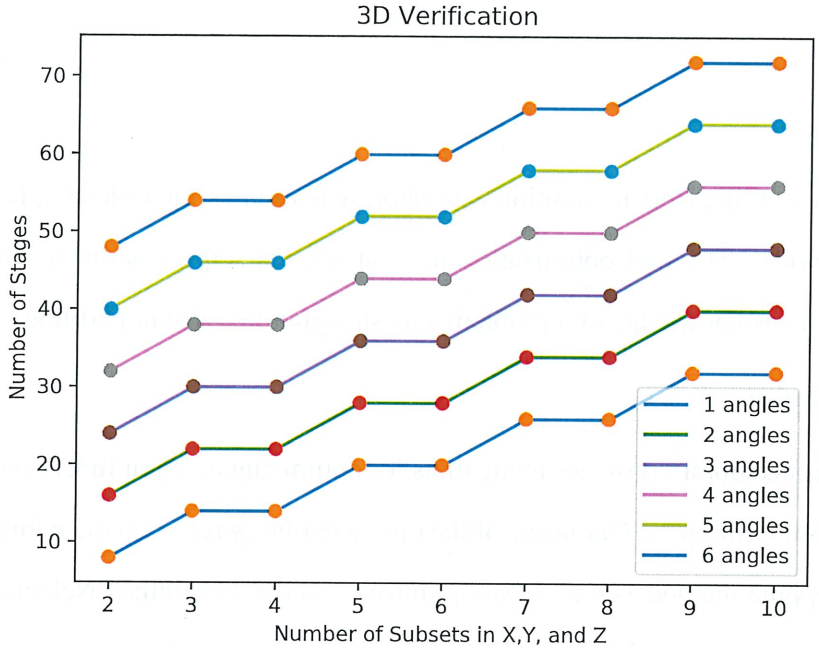


Figure 4.16: A 3D verification suite with regular partitions run from 2^3 to 10^3 subsets with each case being run from 1 to 6 angles per octant.

4.4 PDT performance model vs. TTS

4.5 PDT vs. TTS

I did not read this

5. PARTITION OPTMIZIATION

5.1 Method

With confidence in the time-to-solution estimator, it is used 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.

5.1.1 Scipy optimize

The scipy optimize library 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 method as the global optimizer, and the constrained Nelder-Mead method as the local optimizer.

5.1.1.1 Basinhopping[7]

5.1.1.2 Constrained Nelder-Mead

5.1.2 human optimization

The cdf based optimization method.

5.2 Analytical Mesh Results

Analytical mesh density results will go here.

5.3 Centroid-Mesh Results

PDT mesh results will go here.

5.4 PDT Comparison Results

The PDT comparison goes here.