Partitioning Optimization for Massively Parallel Transport Sweeps on Unstructured Grids

Tarek Ghaddar
Chair: Dr. Jean Ragusa
Committee: Dr. Jim Morel, Dr. Marvin Adams, Dr. Nancy Amato

Nuclear Engineering Department
Texas A&M University
College Station, TX, 77843-3133

Introduction

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 like 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[2] 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.

Review of Neutron Transport and the Transport Sweep

The steady-state neutron transport equation describes the behavior of neutrons in a medium and is given by Eq. (1):

$$\vec{\Omega}\cdot\vec{\nabla}\psi(\vec{r},E,\vec{\Omega})+\Sigma_t(\vec{r},E)\psi(\vec{r},E,\vec{\Omega}) = \int_0^\infty dE'\int_{4\pi} d\Omega'\Sigma_s(\vec{r},E'\to E,\Omega'\to\Omega)\psi(\vec{r},E',\vec{\Omega}')+S_{ext}(\vec{r},E,\vec{\Omega}), \tag{1}$$

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) 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:

$$\begin{aligned}\vec{\Omega}\cdot\vec{\nabla}\psi(\vec{r},E,\vec{\Omega})+\Sigma_t(\vec{r},E)\psi(\vec{r},E,\vec{\Omega}) &= \frac{1}{4\pi}\int_0^\infty dE'\Sigma_s(\vec{r},E'\to E)\int_{4\pi} d\Omega'\psi(\vec{r},E',\vec{\Omega}')+S_{ext}(\vec{r},E,\vec{\Omega})\\ &= \frac{1}{4\pi}\int_0^\infty dE'\Sigma_s(\vec{r},E'\to E)\phi(\vec{r},E')+S_{ext}(\vec{r},E,\vec{\Omega}),\end{aligned} \tag{2}$$

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

$$\phi(\vec{r},E') = \int_{4\pi} d\Omega'\psi(\vec{r},E',\vec{\Omega}'). \tag{3}$$

The next step to solving the transport equation is to discretize in energy, yielding Eq. (4), the multigroup transport equation:

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

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

Next, we discretize in angle using the discrete ordinates method,[3] whereby an angular quadrature $\left(\vec{\Omega}_m, w_m\right)_{1\le m\le M}$ is used to solve the above equations along a given set of directions $\vec{\Omega}_m$:

$$\vec{\Omega}_m\cdot\vec{\nabla}\psi_{g,m}(\vec{r})+\Sigma_{t,g}(\vec{r})\psi_{g,m}(\vec{r})=\frac{1}{4\pi}\sum_{g'}\Sigma_{s,g'\to g}(\vec{r})\phi_{g'}(\vec{r})+S_{ext,g,m}(\vec{r}), \tag{5}$$

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:

$$\phi_g(\vec{r})\approx\sum_{m=1}^{m=M} w_m\psi_{g,m}(\vec{r}). \tag{6}$$

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:

$$\vec{\Omega}_m\cdot\vec{\nabla}\psi_m(\vec{r})+\Sigma_t(\vec{r})\psi_m(\vec{r})=\frac{1}{4\pi}\Sigma_s(\vec{r})\phi(\vec{r})+q_m^{ext+inscat}(\vec{r})=q_m(\vec{r}), \tag{7}$$

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. (8):

$$\vec{\Omega}_m\cdot\vec{\nabla}\psi_m^{(l+1)}(\vec{r})+\Sigma_t\psi_m^{(l+1)}(\vec{r})=q_m^{(l)}(\vec{r}), \tag{8}$$

where the right hand side terms of Eq. (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. (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,[4] finite volume,[4] or discontinuous finite element,[6] 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 shows the sweep ordering for a given direction on both a structured and unstructured mesh.

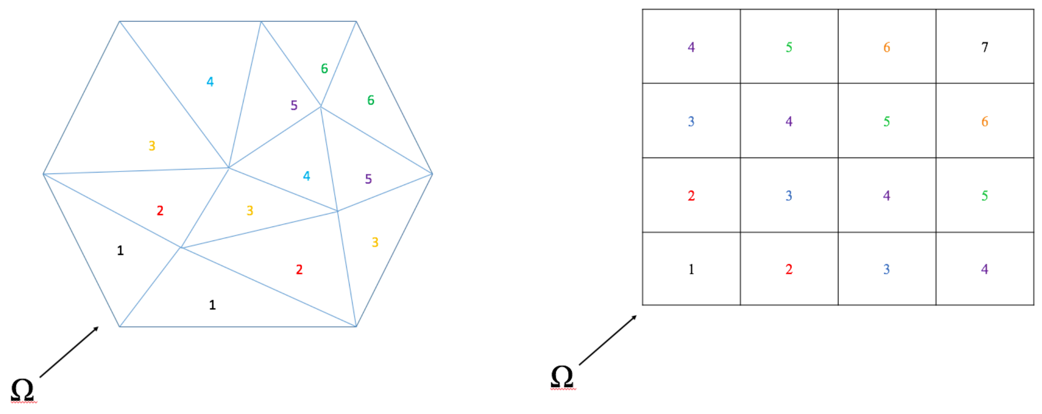

Figure 1. A demonstration of a sweep on a structured and unstructured mesh.

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. 2.

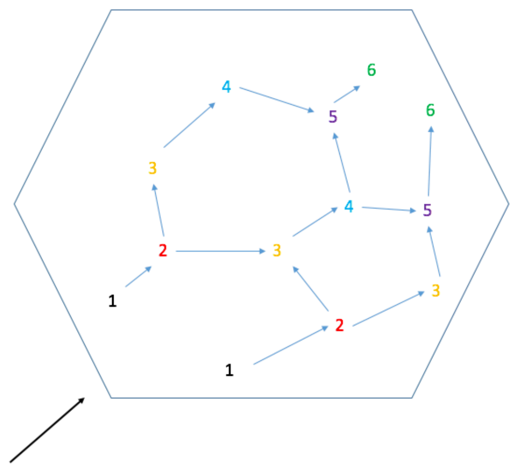

Figure 2. A task dependence graph of the unstructured mesh example in Fig. 1.

The concepts presented in this introduction are used by PDT, Texas A&M University's massively parallel deterministic transport code, capable of multi-group simulations and employing discrete ordinates for angular discretization. PDT features steady-state, time-dependent, criticality, and depletion simulations, and solves the transport equation for neutron, thermal, gamma, coupled neutron-gamma, electron, and coupled electron-photon radiation. PDT has been shown to scale on logically Cartesian grids out to 750,000 cores. Logically Cartesian grids contain regular convex grid units that allow for vertex motion inside them, in order to conform to curved shapes. Much of the work completed and proposed in this document uses PDT or was developed in PDT.

Review of 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. 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[2] :

- 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_xA_y cells.
- $N_g = \frac{G}{A_g}$
- $N_m = \frac{M}{A_m}$
- $N_k = \frac{N_z}{P_zA_z}$
- $N_kA_xA_yA_z = \frac{N_xN_yN_z}{P_xP_yP_z}$

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

One task contains $A_xA_yA_z$ 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 (9) approximately defines parallel sweep efficiency:

$$\begin{aligned} \varepsilon &\approx \frac{T_{\text{task}}N_{\text{tasks}}}{[N_{\text{stages}}][T_{\text{task}} + T_{\text{comm}}]} \\ &\approx \frac{1}{[1 + \frac{N_{\text{idle}}}{N_{\text{tasks}}}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]}, \end{aligned} \tag{9}$$

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 (10) and (11) show how T_{comm} and T_{task} are calculated:

$$T_{\text{comm}} = M_LT_{\text{latency}} + T_{\text{byte}}N_{\text{bytes}} \tag{10}$$

$$T_{\text{task}} = A_xA_yA_zA_mA_gT_{\text{grind}}, \tag{11}$$

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. 11 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,[2] which is an extension of the popular KBA algorithm.[5]

The KBA Algorithm

Several parallel transport sweep codes use KBA partitioning in their sweeping, such as Denovo[3] and PARTISN.[1] 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 3 shows an example of pipelining the angular work of a quadrant.

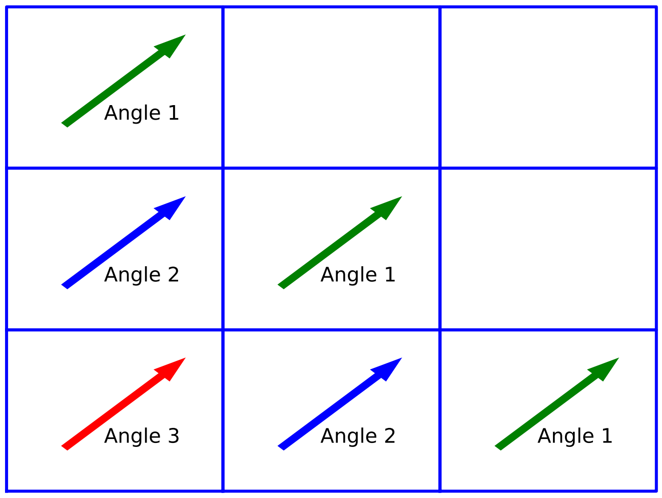

Figure 3. An example showing the pipelining of angular work from the lower left quadrant.

In Fig. 3, 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[2] for "successive in angle, successive in quadrant" is:

$$\varepsilon_{KBA} \approx \frac{1}{[1+\frac{4(P_x+P_y-2)}{8MN_k}][1+\frac{T_{\text{comm}}}{T_{\text{task}}}]}. \tag{12}$$

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.

The PDT optimal parallel efficiency[2] is:

$$\varepsilon \approx \frac{1}{[1+\frac{P_x+P_y-4+N_k(P_z-2}{8MGN_k/(A_m A_G)}][1+\frac{T_{\text{comm}}}{T_{\text{task}}}]} \tag{13}$$

PDT's partitioning and scheduling algorithm has been studied for balanced (equivalent amount of work per processor), regular hexahedral meshes. When PDT introduced support for unstructured meshes, it came at the cost of load-balanced grids. Before discussing how we propose to schedule the transport sweep on such grids, we'll focus on the generation, partitioning, and load balancing of unstructured meshing in PDT.

Review of Partitioning Unstructured Meshes in PDT

The capability for PDT to generate and run on an unstructured mesh gained importance as the need to simulate irregular problems arose. However, there was a desire to keep logically Cartesian partitions in order to preserve as much as the existing sweep mechanics in the code as possible. Each logically Cartesian partition is called a subset, which are obtained using cut planes in 3D and cut lines in 2D. Figure 4 demonstrates this functionality, with the first two subsets meshed using the Triangle Mesh Generator,[7] a 2D mesh generator. PDT employs two approaches to support unstructured meshing, a 2D mesh extruded in the third dimension, and a fully unstructured 3D mesh.

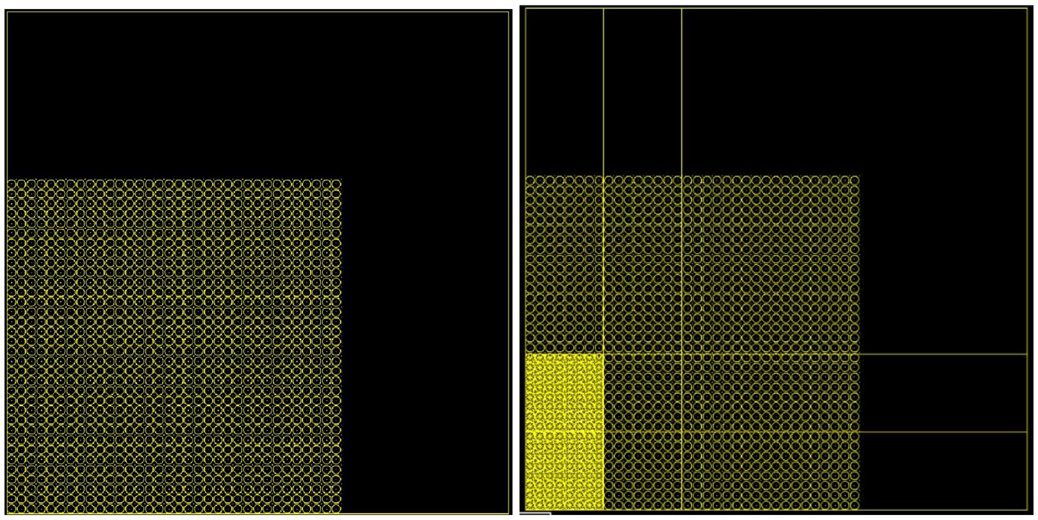

Figure 4. A Planar Straight Line Graph (PSLG) describing a fuel lattice, and with a orthogonal “subset” partition imposed on on the PSLG.

2D Extruded Meshes in PDT

When using the 2D extruded unstructured meshing capability in PDT, the input geometry is described by a Planar Straight Line Graph (PSLG). After superimposing the orthogonal partition, a PSLG is created for each subset, and meshed. Mesh discontinuities along partition boundaries are fixed by “stitching” hanging nodes, creating degenerate polygons. Because PDT’s spatial discretization employs Piece-Wise Linear Discontinuous (PWLD) finite element basis functions, degenerate polygons cause no issues in computing the solution. Because the input’s and each subset’s PSLG must be described and meshed in 2D, the mesh can be extruded in the z dimension in order to give us the capability to run on 3D problems. This method works quite well for symmetric problems, but can cause some issues when meshing spheres, cylinders, and non-symmetric features. As a result, PDT required support for truly 3D unstructured meshes.

3D Unstructured Meshes in PDT

The ability to read in 3D meshes stored in the OpenFOAM format was developed recently in PDT. The OpenFOAM format was chosen for two reasons. First, OpenFOAM has many utilities to convert existing meshes in a variety of formats (such as Fluent) into the OpenFOAM format, which allows PDT to support users with pre-generated meshes. Second, as part of his PhD work, Rick Vega (a Texas A&M graduate under Marvin Adams) wrote a code that takes the OpenFOAM-formatted mesh, partitions it with cut planes (preserving the logically Cartesian partitioning scheme), load balances it, and dumps the partitioned, balanced mesh for PDT to read in.

Figures 5, 6, and 7 show the differences in the mesh between the 2D extruded and 3D approaches for the

IM1C experiment run at Texas A&M University.

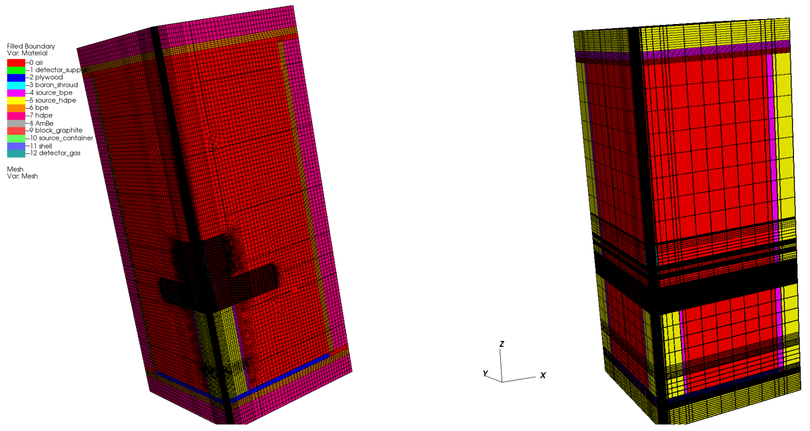

Figure 5. A comparison of the two meshing approaches for the IM1C experiment.

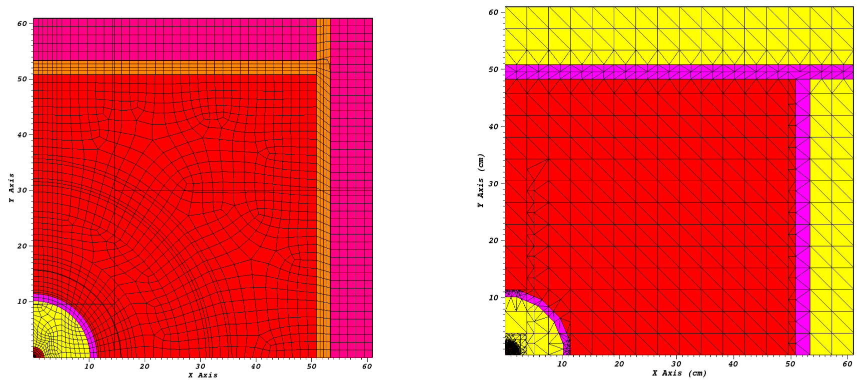

Figure 6. A comparison of the two meshing approaches for a slice into the detector of the IM1C experiment.

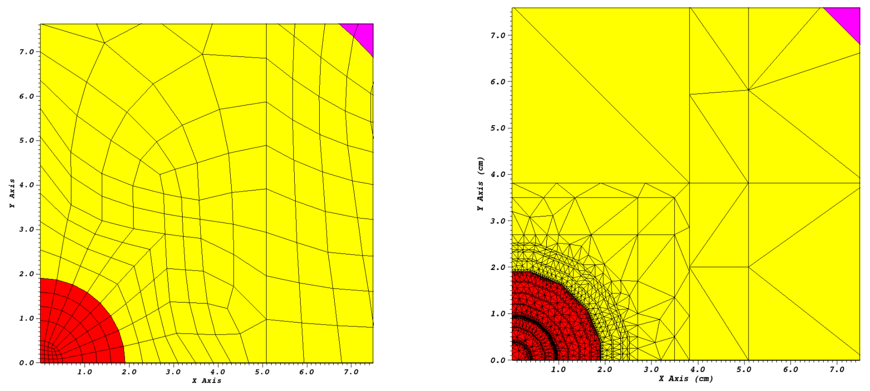

Figure 7. A closer look at the detector region of the two meshing approaches.

As shown in Fig. 7, the other advantage of the OpenFOAM meshes is the ability to maintain good mesh quality with a smaller number of cells. Due to the extrusion process in PDT's prismatic meshes, any spherical or cylindrical shapes are stair-stepped in 2D, and need a lot more cells to maintain a good mesh quality.

However, the 3D meshes do pose a challenge. When aggregating angles for the sweep, cycles occur. This prevents PDT from sweeping on these meshes unless using single angle aggregation, and most sweeps on large problems take far too long to run. A sweep structure that detects and addresses cycles is currently being implemented by the PDT development team, making 3D meshes practical for most problems.

While both unstructured meshing approaches allowed PDT to run problems previously more difficult to run, it introduced the issue of unbalanced partitions. When creating subsets with cut planes or cut lines, there is no guarantee that each subset will have an equivalent amount of cells. Over the last few years, two load balancing algorithms were introduced to combat this issue.

Approaches to Load Balancing Unstructured Meshes

When discussing the parallel scaling of transport sweeps, a load balanced problem is of great importance. A load balanced problem has an equal number of degrees of freedom per processor, and is important in order to minimize idle time for all processors by equally distributing (as much as possible) the work each processor has to do. For the purposes of unstructured meshes in PDT, we are looking to balance the number of cells. Two approaches to load balancing were explored in PDT, which we refer to as "load balancing" and "load balancing by dimension". Both load balancing algorithms rely on the movement of cut planes in order to redistribute high mesh-density areas to multiple processors.

Before describing either load balancing algorithm, we define a metric describing how imbalanced our problem is:

$$f = \frac{\max_{ijk}(N_{ijk})}{\frac{N_{tot}}{I \cdot J \cdot K}}, \tag{14}$$

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:

$$f_D = \max_d [\sum_{d2,d3} N_{ijk}] / \frac{N_{tot}}{D}. \tag{15}$$

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 lines in the respective dimension are redistributed.

Load Balancing

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 initial load balancing algorithm.

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

While the problem is not balanced, we check if the cells per column and cells per row, represented by f_I and f_J, are balanced. If they are not, we redistribute the cut lines. Figure 8 helps illustrate the criteria used to balance the cells per column.

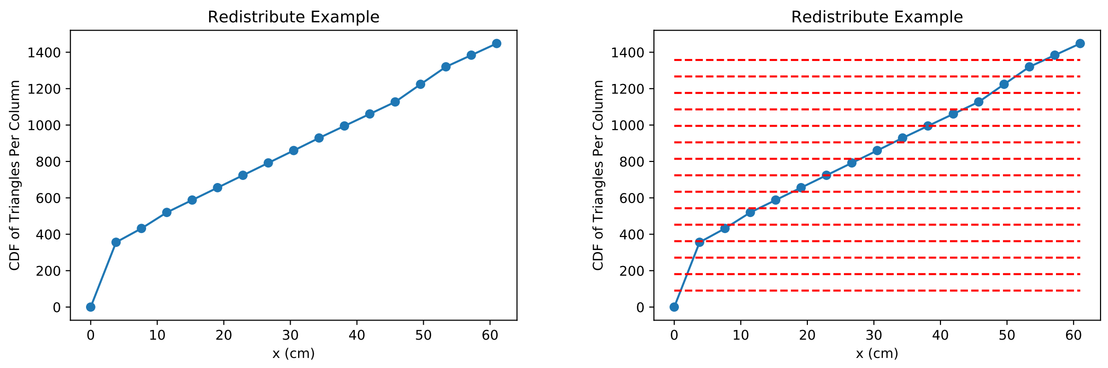

Figure 8. The use of the CDF of triangles per column to redistribute the cut lines in X.

The image on the left side of Fig. 8 shows the CDF of the triangles per column. The red lines on the right side of Fig. 8 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.

Load Balancing by Dimension

An improved load balancing algorithm was implemented that improved our metric. Rather than load balancing all dimensions within an iteration, one dimension is balanced first for all iterations, and then the cut planes that yielded the best metric for that dimension are chosen. Then the next dimension is balanced within the first dimension. For example, if the x cut planes are balanced first, the y cut planes would be balanced *within* each column. When full 3D load balancing by dimension (shown in Fig. 9) is used via Rick Vega's code, excellent metrics (shown in Table I) within 6% of unity are consistently obtained.

Subsets	f
8	1.008
27	1.017
64	1.028
125	1.022
216	1.057

TABLE I
The imbalance factor f for the IM1C experiment for different numbers of subsets after load balancing by dimension.

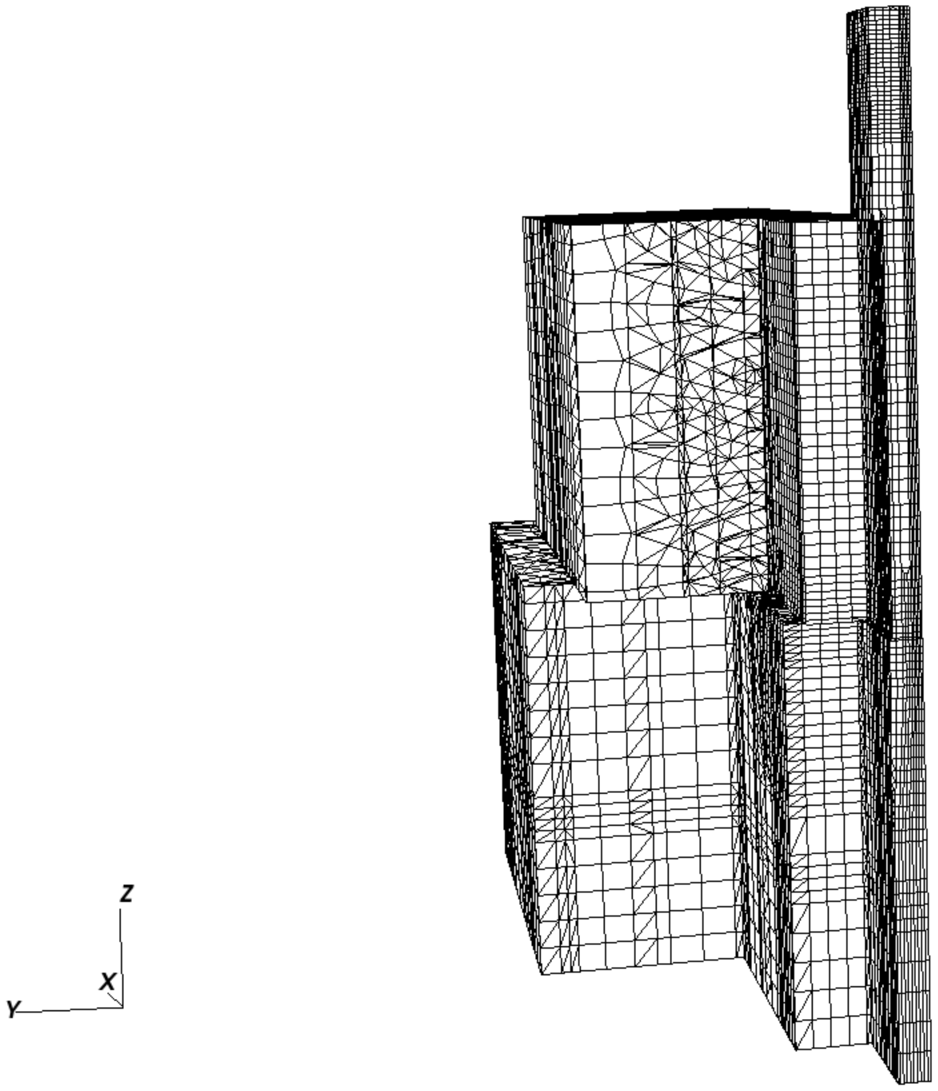

Figure 9. A slice of the IM1C experiment showcasing load balancing by dimension in 3D with an OpenFOAM mesh.

Even with excellent metrics with the load balancing by dimension algorithm, concerns arose that the transport sweep would be impacted by not sweeping across regular partitions (cut planes going all the way across the domain rather than being split by dimension).

Issues of Load Balancing Unstructured Meshes for Transport Sweeps

While perfect load balance was the initial goal, we needed to know if this affected the efficiency of the parallel transport sweep. Figures 10-12 showcase the theoretical effect of ideal load balance on the transport sweep.

Figure 10 shows a sweep domain with a regular partition, $P_x = 20, P_y = 20$, and ideal load balance (equivalent number of cells per processor). The sweep maintains a mostly regular wavelike structure as each angle sweeps through the domain, and this sweep completes in 40 stages. This is the ideal partitioning for the transport sweep, but often hard to achieve with unstructured meshes **and** perfect balance.

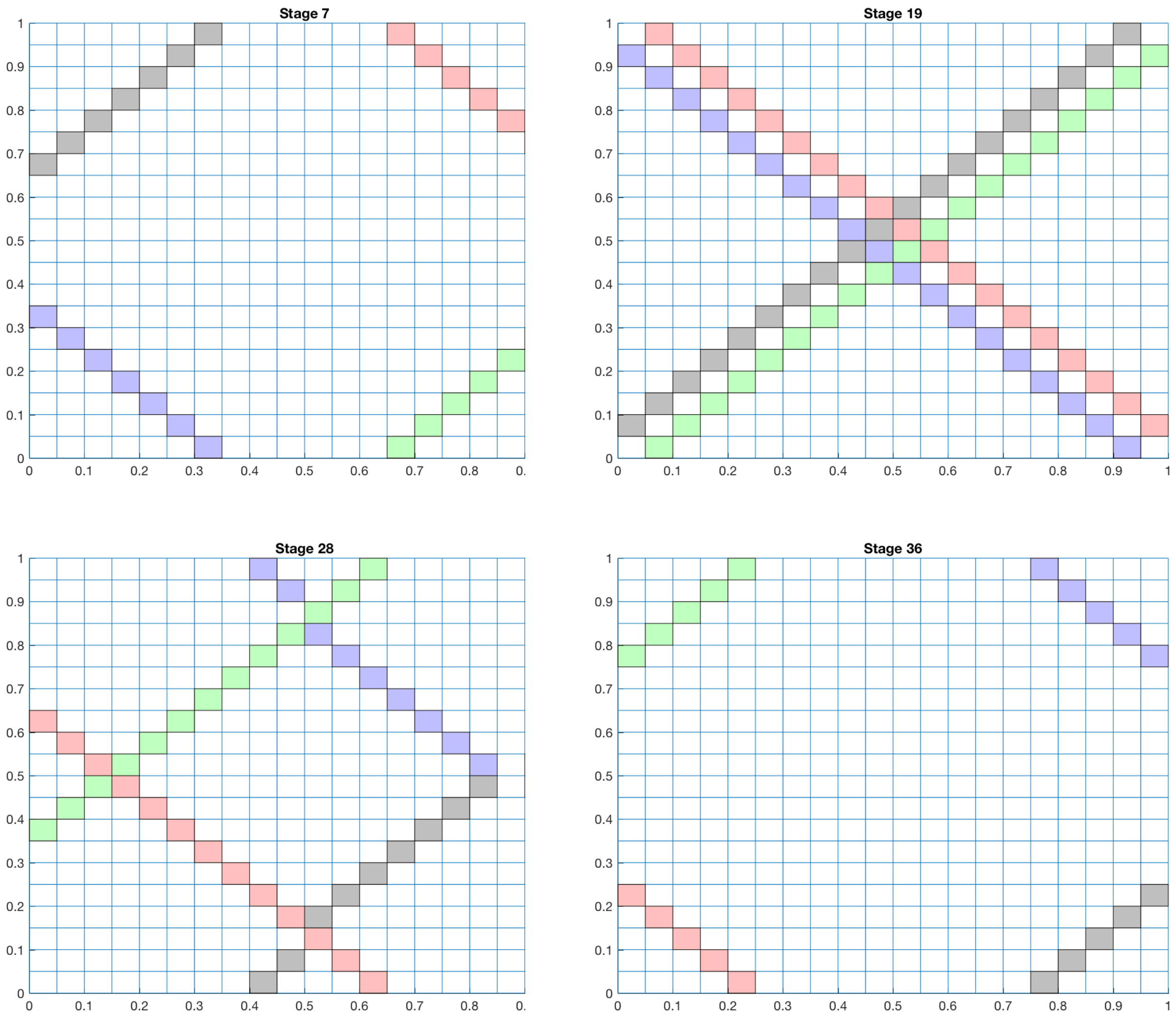

Figure 10. The transport sweep using a regular partition with perfect balance.

With the improved load balancing by dimension, we are likely to see a partition similar to the partitioning shown in Fig. 11 for all geometries. It is immediately observable that we do not have the nice wavelike transport sweep, as communication penalties take their toll. As a result, this sweep completes in 101 stages, as opposed to 40 stages for the regular partition.

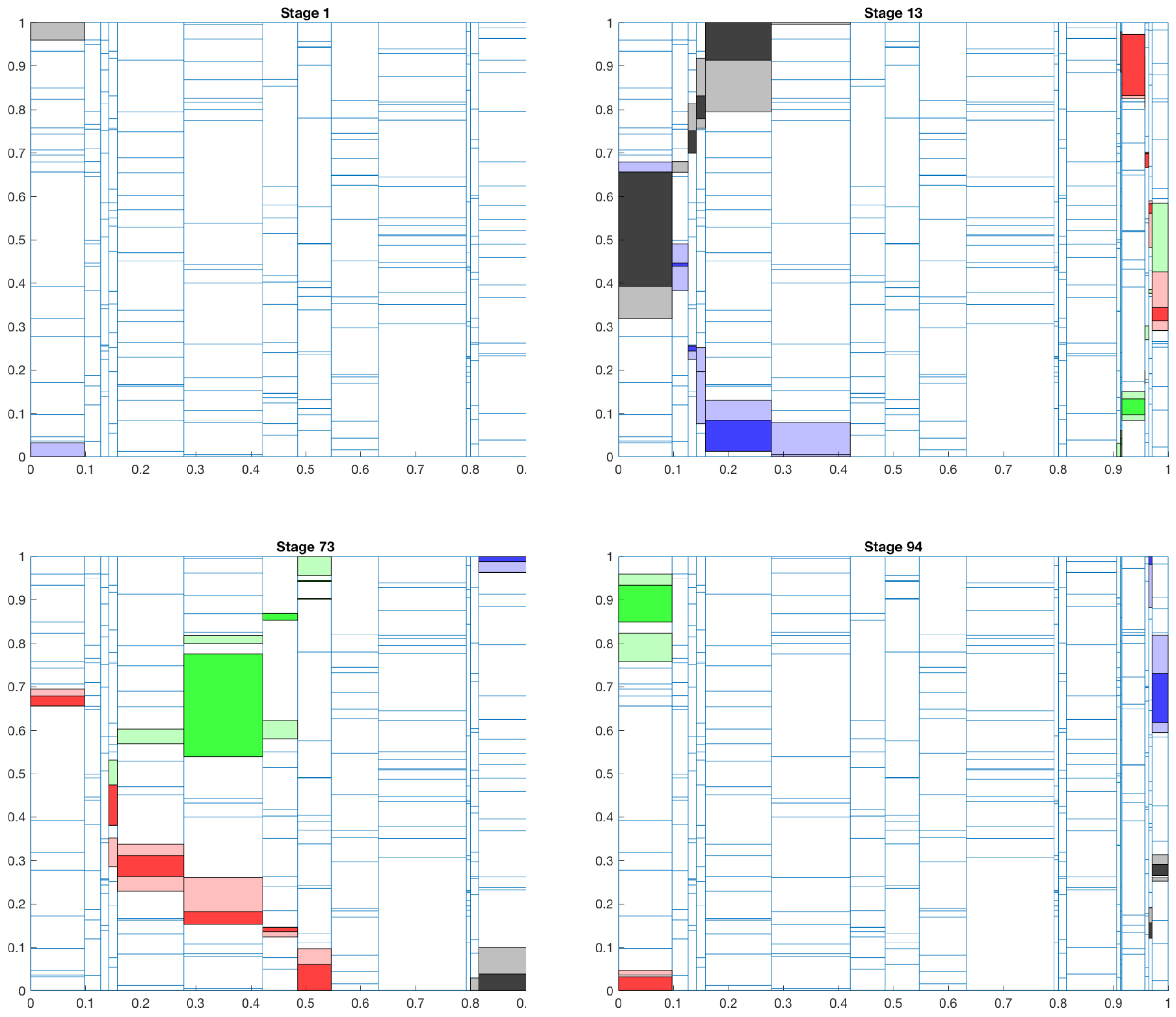

Figure 11. The transport sweep using a random partition with perfect balance.

Figure 12 shows a worse theoretical sweep partitioning, where the transport sweep takes on a snake-like behavior. This partitioning yields a stage count of 230, which is far worse than even the random partitioning stage count of 101.

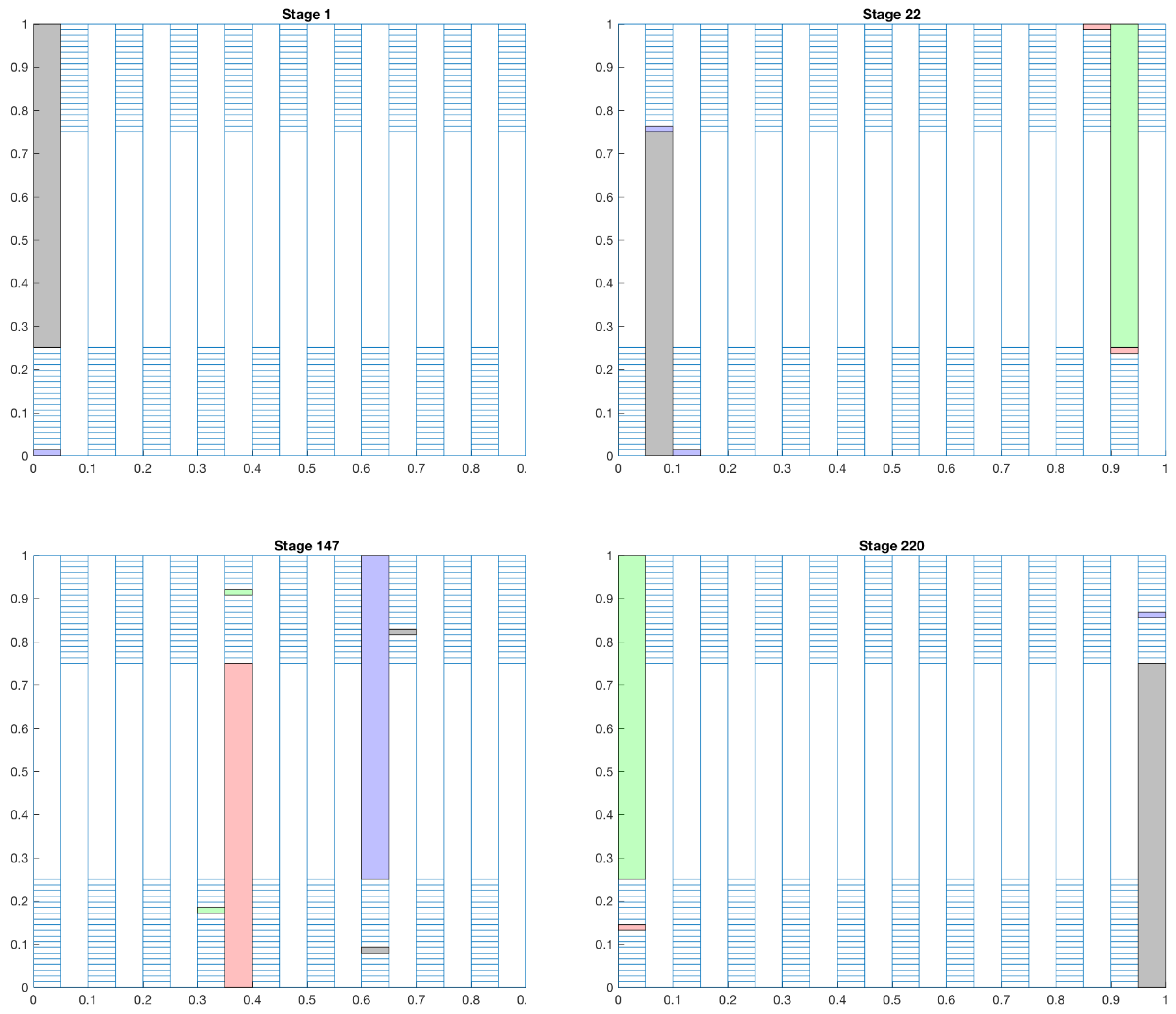

Figure 12. The transport sweep using the worst partition with perfect balance.

Two things become immediately clear as a result of this study: ideal balance does not always yield the best parallel efficiency, and the stage count of a sweep loses importance. This is because the stage relies on the assumption of perfect balance. Once the time to compute a task changes throughout the problem domain, we must instead rely on the total time to solution as the most important metric, not the stage and the previously stated definition of parallel efficiency (Eq. 9).

Time-to-Solution Estimation

In order to optimize the location of the cut planes based on balance and sweep efficiency without relying on a costly iterative process, we must build a time to solution estimator. For a given partition, this tool will estimate the time to solution. This is done by building directed task dependence graphs for each octant (quadrant in 2D) and weighting the graph edges based on certain criteria discussed Section VI.A. Figure 13 shows an example subset partitioning scheme in 3D, and the corresponding task dependence graph (TDG) for the front bottom octant is shown in Fig. 14.

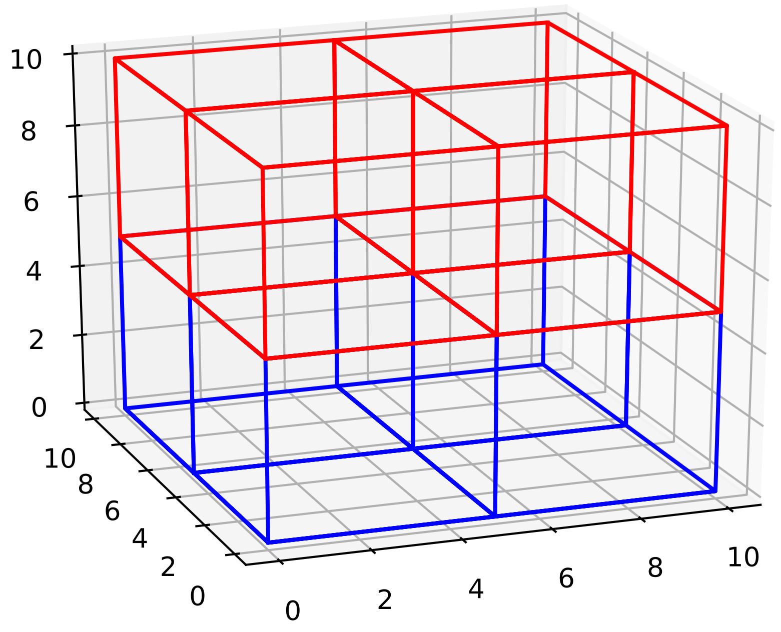

Figure 13. An example uniform partitioning scheme.

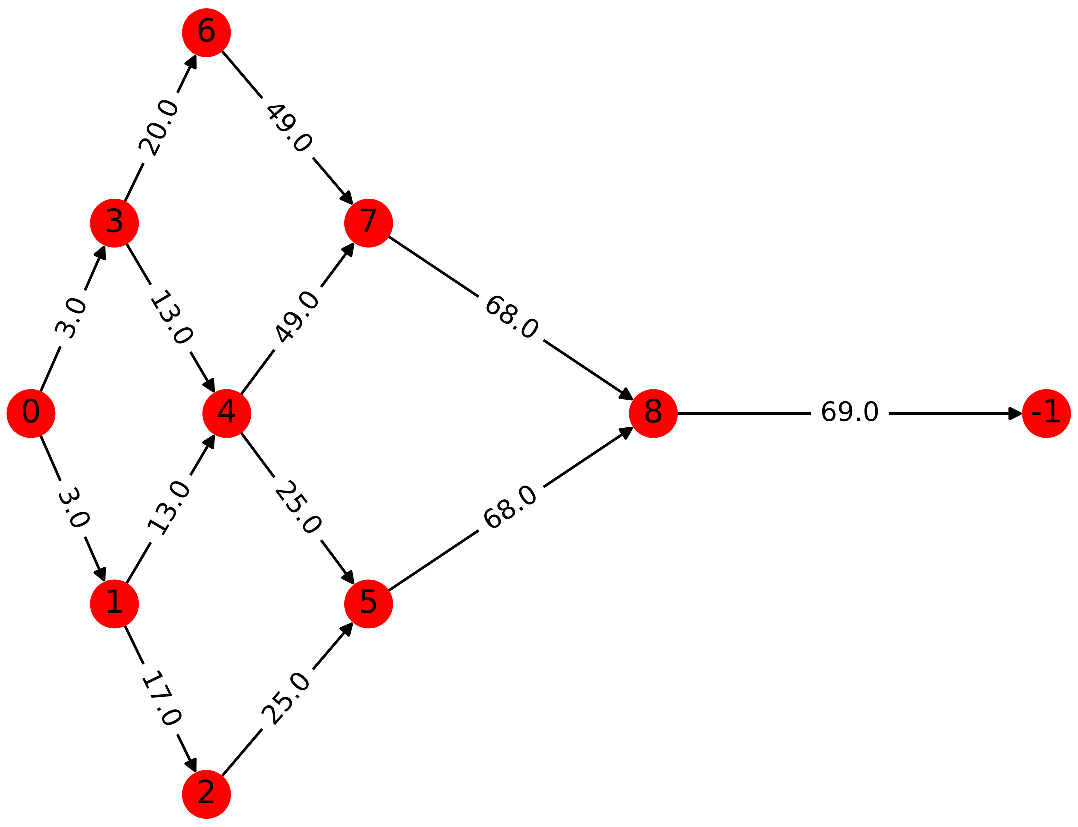

Figure 14. The task dependence graph for the front bottom octant in Fig. 13 with weights assigned.

Weighting the Task Dependence Graph

The time to solution relies on weighting the graph correctly to determine the traversal time for each octant. In order to correctly estimate the time to solution, two pieces of information are initially required: the number of cells per subset, and the number of boundary cells per subset. In order to obtain this information, a very fine grid is superimposed upon the domain partitioning, with each fine grid unit containing g cells. Equation (16) gives the total number of fine grid units n_s, and Eqns. (17) - (19) give $a, b,$ and c, the global domain boundaries in each dimension.

$$n_s = \frac{\text{number of cells across the entire domain}}{g} \tag{16}$$

$$a = x_{\text{max}} - x_{\text{min}} \tag{17}$$

$$b = y_{\text{max}} - y_{\text{min}} \tag{18}$$

$$c = z_{\text{max}} - z_{\text{min}} \tag{19}$$

Equations (20) - (22) calculate the number of fine grid units in each direction.

$$n_x = a \cdot \left(\frac{n_s}{a \cdot b \cdot c}\right)^{1/3} \tag{20}$$

$$n_y = b \cdot \left(\frac{n_s}{a \cdot b \cdot c}\right)^{1/3} \tag{21}$$

$$n_z = c \cdot \left(\frac{n_s}{a \cdot b \cdot c}\right)^{1/3} \tag{22}$$

Combining $n_x, n_y,$ and n_z with the cut line information in each direction allows us to estimate the fine grid units in each subset and along each boundary.

Given that we know the machine parameters (grind time, latency, message communication time), Eq. 23 estimates how long it takes to solve and communicate all unknowns in a given subset:

$$\text{cost} = N_s \cdot T_g + N_b \cdot T_{\text{comm}} + \text{latency} \cdot M_l, \tag{23}$$

where N_s is the number of unknowns in the subset, T_g is the grind time, N_b is the number of unknowns along the subset boundaries, T_{comm} is the communication time per unknown, and M_l is the machine-dependent latency multiplier.

The cost value is used to help appropriately weight the edges in each TDG independently. The outgoing edges of each node are weighted with this cost value, representing the time it takes to solve and communicate information to neighboring nodes. Once this is done, all incoming edges to each node are set to the value of the longest path to the node. Every incoming edge now represents the time each node is ready to solve, taking into account upstream dependencies. Algorithm 2 summarizes this process:

Algorithm 2 Weighting the TDGs.

```
for $node = 1$ to $N$ do
    Weight outgoing edges with the cost of solving and communicating $node$.
end for
for $node = 1$ to $N$ do
    Find longest path to $node$
    Set all incoming edge weights to $node$ to the sum of the longest path to the node.
end for
```

Now that each TDG is independently weighted, we need to address conflicts across all TDGs. This will also be done by modifying the weights of each TDG to reflect these conflicts.

Conflict Detection and Resolution

The conflict detection process is best explained as a “marching” process. We begin at time $t = 0$, and march along each TDG until one of the TDGs reaches a node. Recall that the incoming weight to a node reflects the time t that it is ready to solve. If at time t, multiple TDGs are solving the same node, this means we have a conflict. Whichever TDGs lose the conflict modify their downstream weights according to how long they are delayed. Algorithm summarizes conflict detection and resolution.

Algorithm 3 The conflict detection and resolution algorithm.

```
t = 0
while num_finished_graphs < num_graphs do
    Get nodes that are being solved at time t
    Find conflicting graphs in nodes being solved
    if No conflicting graphs in nodes being solved then
        t = time of next interaction
    else
        Find first conflicted node
        Get conflicting graphs at first conflicted node
        Modify graph weights according to which graph wins the conflict
        t = time of next interaction
    end if
    Check how many graphs have finished sweeping by time t
end while
```

Due to the potentially imbalanced nature of problems, each subset could take a different amount of time to solve and communicate its information. Conflicts are resolved by one of two methods, first-come-first-serve and depth-of-graph remaining. If a conflict arises between two or more TDGs, the following process is followed:

1. Whichever TDG reaches the conflicted node first, wins.
2. If two or more TDGs reach the conflicted node at the same time, the TDG with the greater depth-of-graph remaining wins.
3. If two or more TDGs have the same depth-of-graph remaining, the graph with the priority octant wins (the same as PDT’s priority octant system described in Section II.B.).

Once all TDGs have had their weights modified to address conflicts, it is extremely simple to estimate the time-to-solution. The maximum value of the final weight (for example, the weight from node 8 to node -1 in Fig. 14) across all TDGs is the time-to-solution.

Optimizing the Time to Solution

Once each TDG is weighted appropriately with conflicts taken into account, we simply take the final weight of the ending node as the time to solution for that graph. The octant with the heaviest sweep time is used as the time to solution for a given partitioning scheme. We will use a python optimization tool in order to get the best partitioning scheme possible. The cut planes will be the input parameter, and the tool will iterate on this parameter to minimize the time to solution.

Goals and Proposed Plan

- Devise and implement a load balancing by dimension algorithm in PDT (Section IV.B.).
- Devise and implement a time-to-solution estimator, with angle-set and cell-set aggregation capabilities, that returns the maximum time to solution for a given partitioning scheme (Section VI.).
- Verify time-to-solution estimator matches PDT's sweep time with equivalent aggregation parameters and scheduling.
- Use time-to-solution estimator to optimize cut plane locations by using it as a cost function in a Python optimization library (scipy.minimize, scipy.optimize, etc.).
- Build test problems, in 2D and 3D, that balance approximately perfectly via both load balancing algorithms.
- Using these test problems, show results obtained by optimizing partitions with the proposed approach.

References

[1] RE Alcouffe, RS Baker, SA Turner, and JA Dahl. Partisn manual. Technical report, LA-UR-02-5633, Los Alamos National Laboratory, 2002.

[2] 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*, 2013.

[3] Thomas M. Evans et al. Denovo: A new three-dimensional parallel discrete ordinates code in scale. *Nuclear Technology*, 171:171–200, 2010.

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

[5] Randal S. Baker & Kenneth R. Koch. An s_n algorithm for the massively parallel cm-200 computer. *Nuclear Science and Engineering*, 128, March 1998.

[6] W. H. Reed and T. R. Hill. Triangular mesh methods for the neutron transport equation. *Los Alamos National Laboratory Publications*, LA-UR-73-379.

[7] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.