M&C 2017 - International Conference on Mathematics & Computational Methods Applied to Nuclear Science & Engineering, Jeju, Korea, April 16-20, 2017, on USB (2017)

An Approach for Load Balancing Massively Parallel Transport Sweeps on Unstructured Grids

Tarek H. Ghaddar, Jean C. Ragusa

Dept. of Nuclear Engineering, Texas A&M University, College Station, TX, 77843-3133
tghaddar@tamu.edu, jean.ragusa@tamu.edu

Abstract - *Load balancing refers to the practice of distributing approximately equal amounts of work among processors so that all processors are kept busy all of the time. When running massively parallel codes, load balancing is a priority in order to improve parallel efficiency. Load balancing is important in order to minimize idle time for all processors by equally distributing (as much as possible) the work for each processor. Here, we discuss load-balanced problems in terms of an (approximately) equal number of degrees of freedom per processor, provided that we are employing a partitioning scheme that minimizes idle time. Recently , an unstructured meshing capability was implemented in PDT, Texas A&M University's massively parallel deterministic transport code, hence allowing the user to define more realistic problem geometries and to define 3D problems through the extrusion of 2D meshes. However, unstructured grids are significantly harder to load balance than logically-Cartesian meshes. In this work, we propose a load balancing algorithm that is now implemented in PDT in order to minimize a specific load-imbalance metric. Several numerical test cases are provided.*

I. INTRODUCTION

When running a massively parallel code, attaining load balancing is a priority in order to improve parallel efficiency. A load balanced problem has an equal number of degrees of freedom per processor. This is attained by equally distributing (as much as possible) the work load amongst the processors so that they are all kept busy at all times.

To the best of our knowledge, the transport sweep is the only scalable solution technique on the current leadership class supercomputers. PDT, Texas A&M University's massively parallel deterministic transport code, has been shown to scale on logically Cartesian grids out to 768,000 cores [1]. Logically Cartesian grids are constructed with mesh cells that are identified using integer triplets ijk (i.e., cubic cells), but allow for vertex motion in order to conform to curved shapes. PDT solves radiation transport problems (neutron, gamma, coupled neutron-gamma, electron, coupled electron-photon, and radiative transfer). It uses discrete ordinates for angular discretization[2, 3], multi-group and FEDS[4] energy differencing, and discontinuous finite elements in space.

A new unstructured meshing capability was implemented in PDT in order to realistically represent certain geometries. Cut lines (cut planes for 3D cases) are used to partition such geometries into logically-Cartesian subdomains, which are then individually meshed in parallel using the Triangle Mesh Generator [5]. These subdomains are then "stitched" or "glued" together in order to create a continuous geometry. 2D meshes can be extruded in the z dimension for 3D problems.

However, unstructured meshes often create unbalanced problems due to the way localized features are meshed, so a load balancing algorithm was added into PDT.

II. THEORY

1. The Transport Equation

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

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

where $\boldsymbol{\Omega}\cdot\boldsymbol{\nabla}\psi$ is the leakage term and $\Sigma_t\psi$ is the total collision term (absorption, outscatter, and within group scattering). The right hand side of Eq. (1) represents the gain terms, where S_{ext} is the external source of neutrons and $\int_0^\infty dE'\int_{4\pi}d\Omega'\Sigma_s(E'\rightarrow E,\Omega'\rightarrow\Omega)\psi(\boldsymbol{r},E',\boldsymbol{\Omega}')$ is the inscatter term, which represents all neutrons scattering from energy E' and direction $\boldsymbol{\Omega}'$ into energies about E and directions about $\boldsymbol{\Omega}$. We assumed a non-multiplying medium and will further assume isotropic scattering for simplicity and conciseness. We introduce the scalar flux as the integral of the angular flux:

$$\phi(\boldsymbol{r},E')=\int_{4\pi}d\Omega'\psi(\boldsymbol{r},E',\boldsymbol{\Omega}').\quad(2)$$

Using the multigroup approximation yields:

$$\begin{aligned}&\boldsymbol{\Omega}\cdot\boldsymbol{\nabla}\psi_g(\boldsymbol{r},\boldsymbol{\Omega})+\Sigma_{t,g}(\boldsymbol{r})\psi_g(\boldsymbol{r},\boldsymbol{\Omega})\\ &=\frac{1}{4\pi}\sum_{g'}\Sigma_{s,g'\rightarrow g}(\boldsymbol{r})\phi_{g'}(\boldsymbol{r})+S_{ext,g}(\boldsymbol{r},\boldsymbol{\Omega}),\quad\text{for } 1\le g\le G\end{aligned}\quad(3)$$

where the multigroup transport equations now form a system of G coupled equations. Next, we discretize in angle using the discrete ordinates method, whereby an angular quadrature

$(\mathbf{\Omega}_m, w_m)_{1\leq m\leq M}$ is used to solve the above equations along a given set of directions $\mathbf{\Omega}_m$:

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

where the subscript m is introduced to describe the angular flux in direction m. The scalar flux integral is numerically evaluated

$$\phi_g(\boldsymbol{r}) \approx \sum_{m=1}^{m=M} w_m\psi_{g,m}(\boldsymbol{r}). \quad (5)$$

From Equation (3), it is clear that we are solving a sequence of transport equations, one equation per group and per direction. Therefore, all transport equations are of the following form:

$$\mathbf{\Omega}_m\cdot\nabla\psi_m(\boldsymbol{r})+\Sigma_t(\boldsymbol{r})\psi_m(\boldsymbol{r}) = \frac{1}{4\pi}\Sigma_s(\boldsymbol{r})\phi(\boldsymbol{r})+q_m^{ext+inscat}(\boldsymbol{r}) = q_m(\boldsymbol{r}), \quad (6)$$

where the group index notation is omitted for brevity. In order to obtain the solution for this discrete form of the transport equation, source iteration is introduced, for instance.

$$\mathbf{\Omega}_m \cdot \nabla\psi_m^{(l+1)}(\boldsymbol{r}) + \Sigma_t\psi_m^{(l+1)}(\boldsymbol{r}) = q_m^{(l)}(\boldsymbol{r}), \quad (7)$$

where the right hand side terms of Eq. (4) 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. (7) must be discretized in space as well. We use a discontinuous Galerkin approximation in space, and 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. The solution is obtained by meshing the domain and solving the spatial problem one cell at a time for a given direction and a given group. Figure 1 shows the sweep ordering for a given direction on both 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 solving for their own solution. This dependency can be represented and stored as a directed task dependence graph, shown in Fig. 2.

The order in which radiation in a cell is solved is given by a task dependence graph. Transport sweep can be performed on parallel architectures in order to obtain the solution faster, as well as distribute the memory to many processors for memory intensive cases. Provably-optimal transport sweeping has been described in [6] and demonstrated in PDT using logically-Cartesian meshes. Our work utilizes that transport sweep machinery but adapts it to unstructured meshes. Performing a transport sweep on an unstructured mesh presents two challenges: (1) performing a transport sweep on a massively parallel scale in an efficient manner and (2) keeping non-concave sub-domains due to leverage fromt he provably-optimal transport sweep algorithms.

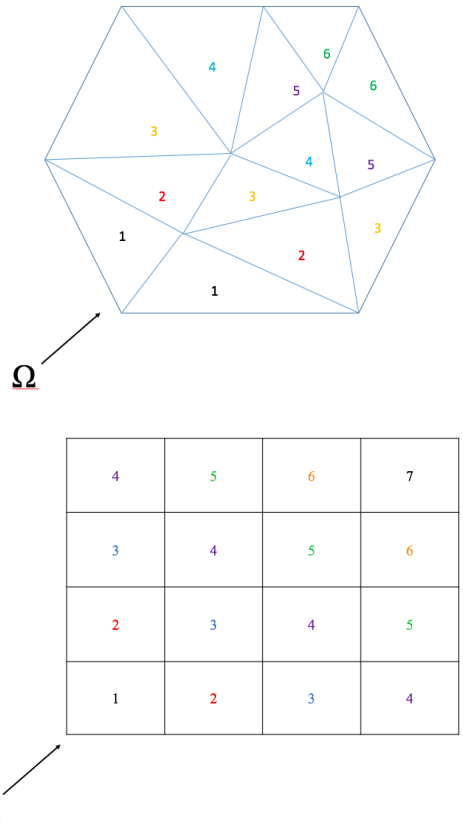

Fig. 1. A demonstration of a sweep on a structured and unstructured mesh.

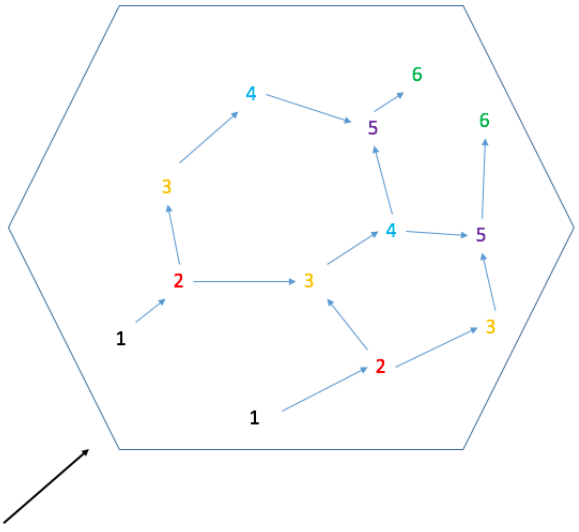

Fig. 2. A task dependence graph of the unstructured mesh example in Fig. 1.

2. The Parallel Transport Sweep

A parallel sweep algorithm is defined by three properties [6] :

- 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. A structured transport sweep takes place on a Cartesian mesh. Furthermore, the work proposed utilizes aspects of the structured transport sweep.

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 groups unknowns in a single cell, or $\psi_{m,g}$ for a single cell.

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". If M is the total number of angular directions, 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 groups 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
- $N_g = \frac{G}{A_g}$
- $N_m = \frac{M}{A_m}$
- $N_k = \frac{N_z}{P_z A_z}$
- $N_k A_x A_y A_z = \frac{N_x N_y N_z}{P_x P_y P_z}$

It follows that each process owns N_k cell-sets (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. One task takes a stage to complete. This is particularly important when comparing sweeps to the performance models.

Equation (8) approximately defines parallel sweep efficiency. This can be calculated for specific machinery and partitioning parameters by substituting in values calculated using Eqs. (12), (13), and (14).

$$\begin{aligned} \epsilon &= \frac{T_{\text{task}} N_{\text{tasks}}}{[N_{\text{stages}}][T_{\text{task}} + T_{\text{comm}}]} \\ &= \frac{1}{[1 + \frac{N_{\text{idle}}}{N_{\text{tasks}}}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]} \end{aligned} \tag{8}$$

Equations (9) and 10 show how T_{comm} and T_{task} are calculated:

$$T_{\text{comm}} = M_L T_{\text{latency}} + T_{\text{byte}} N_{\text{bytes}} \tag{9}$$

$$T_{\text{task}} = A_x A_y A_z A_m A_g T_{\text{grind}} \tag{10}$$

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. If a high value of M_L is necessary for the performance model to match computational results, improvements should be made in code implementation.

A. KBA Partitioning for Structured Grids

Several parallel transport sweep codes use KBA partitioning in their sweeping, such as Denovo [7] and PARTISN [8]. The KBA partitioning scheme and algorithm was developed by Koch, Baker, and Alcouffe [8].

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. With $N_k = N_z/A_z$, each processor performs $N_{\text{tasks}} = 8MN_k$ tasks. With the KBA algorithm, $2MN_k$ tasks are pipelined from a given corner of the 2D processor layout. The far corner processor remains idle for the first $P_x + P_y - 2$ stages, which means that an octant-pair (or quadrant) sweep completes in $2MN_k + P_x + P_y - 2$ stages. If an octant-pair sweep does not begin until the previous pair's finishes, the full sweep requires $8MN_k + 4(P_x + P_y - 2)$ stages, which means the KBA parallel efficiency is:

$$\varepsilon_{KBA} = \frac{1}{[1 + \frac{4(P_x+P_y-2)}{8MN_k}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]} \tag{11}$$

B. The Structured Transport Sweep in PDT

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 (12), (13), and (14) define N_{fill}, N_{idle}, and N_{tasks}:

$$N_{\text{fill}} = \frac{P_x + \delta_x}{2} - 1 + \frac{P_y + \delta_y}{2} - 1 + N_k(\frac{P_z + \delta_z}{2} - 1) \tag{12}$$

$$N_{\text{idle}} = 2N_{\text{fill}} \tag{13}$$

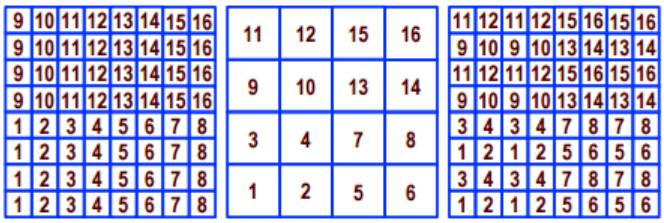

Fig. 3. Three different partitioning schemes in 2D, from left to right: KBA, volumetric non-overloaded, and volumetric overloaded [1].

$$N_{\text{tasks}} = 8N_mN_gN_k \tag{14}$$

where δ_u is 1 for P_u odd, and 0 for P_u even.

When using KBA, P_z is fixed to 1, and with hybrid KBA, P_z is fixed to 2. Volumetric partitioning means that P_z is greater than two. Figure 3 shows three different partitioning schemes used in transport sweeps, KBA (which is defined in the previous section), volumetric non-overloaded, and volumetric overloaded. Volumetric non-overloaded requires that all cells owned by a processor are contiguous, where as volumetric non-overloaded partitioning does not have this restriction.

For a thorough and complete description of partitioning, aggregation, and scheduling in PDT, please refer to [1].

C. The Unstructured Transport Sweep

In an unstructured mesh, the number of cells cannot be described in the same way as a structured mesh. In PDT, the 2D geometry is first subdivided into subsets, which are just rectangular subdomains. Within each subset, an unstructured mesh is generated (using Triangle) and then extruded in 3D. These subsets become the N_x, N_y, N_z equivalent for an structured mesh. The spatial aggregation in a PDT unstructured mesh is done by aggregating subsets into cellsets.

While the PDT transport sweep on structured grids has scaled well out to 768,000 cores, similar levels of parallel scaling have not been achieved using unstructured sweeps yet. Further work is being done to determine how well PDT scales on unstructured meshes. The load balancing algorithm described in this paper was the first step to scaling on unstructured meshes.

III. LOAD BALANCING METHOD

The capability for PDT to generate and run on an unstructured mesh is important because it allows us to run problems without having to conform our mesh to the problem as much. The idea is to have a logically Cartesian grid (creating orthogonal “subsets”) with an unstructured mesh inside each subset. These logically Cartesian subdomains are obtained using cut planes in 3D and cut lines in 2D. Figure 4 demonstrates this functionality. It is decomposed into 3 subsets in x and 3 in y, with the first two subsets meshed using the Triangle Mesh Generator[5], a 2D mesh generator.

This orthogonal grid is superimposed and each subset is meshed in parallel. Subsets are now the base structured unit when calculating our parallel efficiency. Discontinuities along the boundary are fixed by “stitching” hanging nodes, creat-

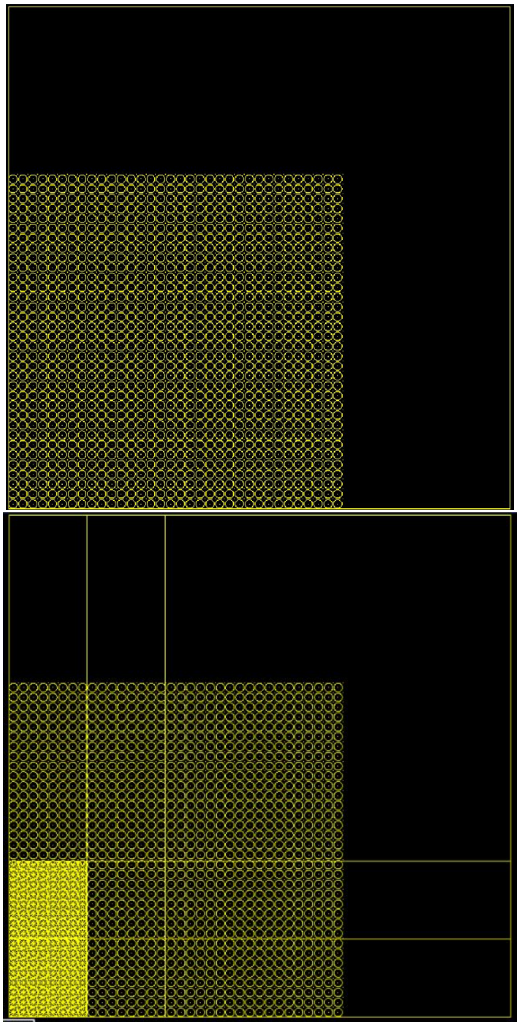

Fig. 4. A PSLG describing a fuel lattice, and with an orthogonal “subset" grid imposed on the PSLG.

ing degenerate polygons along subset boundaries. Because PDT's spatial discretization employs Piece-Wise Linear Discontinuous (PWLD) finite element basis functions, there is no problem solving on degenerate polygons.

When using the unstructured meshing capability in PDT, the input geometry is described by a Planar Straight Line Graph (PSLG)[5]. After superimposing the orthogonal grid, a PSLG is created for each subset, and meshed. 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. Obviously, this is not as good as general an unstructured grid as a tetrahedral grid, but for many problems (e.g., reactor problems), it is a useful capability to have and it can be employed to assess load-balancing algorithm.

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. Load balancing is important in order to minimize idle time for all processes by equally distributing (as much as possible) the work on each process. For the purposes of unstructured meshes in PDT, we are looking to "balance" the number of cells. Ideally, each process will be responsible for an equal number of cells.

If the number of cells in each subset can be reasonably balanced, then the problem is effectively load balanced. The Load Balance algorithm described below details how the subsets will be load balanced. In summary, the procedure of the algorithm involves moving the initially user specified x and y cut planes, re-meshing, and iterating until a reasonably load balanced problem is obtained. Equation 15 shows the equation for calculating the load balancing metric, which dictates how balanced or unbalanced the problem is.

$$f = \frac{\max_{ij}(N_{ij})}{\frac{N_{tot}}{I \cdot J}}, \tag{15}$$

where f is the load balance metric, N_{ij} is the number of cells in subset i, j, N_{tot} is the global number of cells in the problem, and I and J are the total number of in the x and y direction, respectively. The metric is a measure of the maximum number of cells per subset divided by the average number of cells per subset.

The load balancing algorithm moves cut planes based on two sub-metrics, f_I and f_J. Equation (16) defines these two parameters:

$$\begin{aligned} f_I &= \max_i [\sum_j N_{ij}] / \frac{N_{tot}}{I} \\ f_J &= \max_j [\sum_i N_{ij}] / \frac{N_{tot}}{J}. \end{aligned} \tag{16}$$

f_I is calculated by taking the maximum number of cells per column and dividing it by the average number of cells per column. f_j is calculated by taking the maximum number of cells per row and dividing it by the average number of cells per row. If these two numbers are greater than predefined tolerances, the cut lines in the respective directions are redistributed. Once redistribution and remeshing occur, a new metric is calculated. This iterative process occurs until a maximum number of iterations is reached, or until f converges within the user defined tolerance. The Load Balance algorithm behaves as follows:

```
//I,J subsets specified by user
//Check if all subsets meet the tolerance
while (f > tol_subset)
{
  //Mesh all subsets
  if (f_I > tol_column)
  {
    Redistribute(X);
  }
  if (f_J > tol_row)
  {
    Redistribute(Y);
  }
}
//Remesh to get the final mesh.
```

Redistribute: A function that moves cut lines in either X or Y.

Input: CutLines (X or Y vector that stores cut lines).
Input: num_tri_row or num_tri_col, # of tri in each row/col
Input: The total number of triangles in the domain, N_{tot}
stapl::array_view num_tri_view, over num_tri_row/column
stapl::array_vew offset_view
stapl::partial_sum(num_tri_view) {Perform prefix sum}
{We now have a cumulative distribution stored in offset_view}
for i = 1 :CutLines.size()-1 **do**
  vector <double> pt1 = [CutLines(i-1), offset_view(i-1)]
  vector <double> pt2 = [CutLines(i), offset_view(i)]
  ideal_value = $i \cdot \frac{N_{tot}}{\text{CutLines.size()}-1}$
  {Calculate X-intersect of the line formed by pt1 and pt2}
  {and the line y = ideal_value.}
  X-intersect(pt1,pt2,ideal_value)
  CutLines(i) = X-intersect
end for

IV. RESULTS AND ANALYSIS

The following sections will showcase the metric behavior and convergence for three test cases, solution verification for pure absorber and pure scatterer 2D slab problems, and the new unstructured meshing capability both in 2D and 3D.

1. Description of Test Cases

In order to showcase the behavior of the load balancing metric, calculated by Eq. 15 three test cases are presented. Figure 5 shows the first test case, a 20 cm by 20 cm domain with two pins in opposite corners of the domain. Figure 6 shows the same size domain but with the pins on the same side.These are two theoretically very unbalanced cases, as geometrically there are two features located distantly from each other with an empty geometry throughout the rest of the domain. Figure 7 shows a lattice and reflector, which due to it's denser and repeated geometry, theoretically is a more balanced problem.

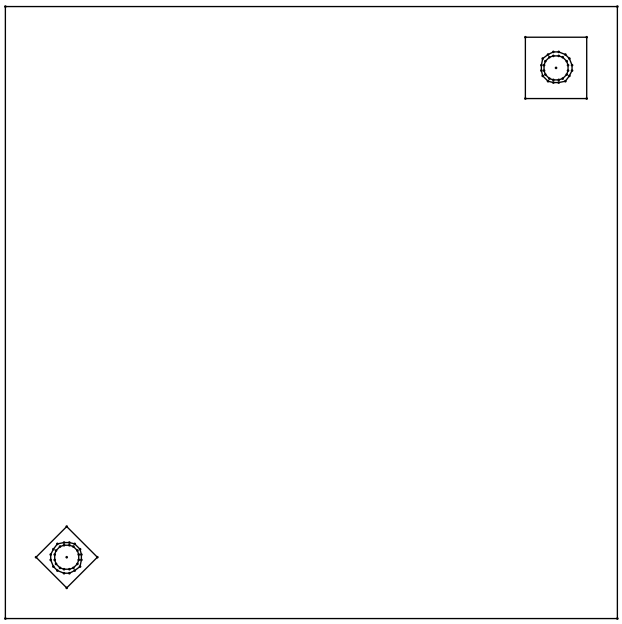

Fig. 5. The first test case used in order to test effectiveness and convergence of the load balancing metric.

A series of 162 inputs was constructed for each case. These inputs are constructed by varying the maximum triangle area from the coarsest possible (refers to Triangle utilizing the fewest possible triangles to mesh the geometry) to 0.01 cm^2 and the number of subsets, N from 2×2 to 10×10.

2. Load Balancing Analysis

For each test case, the 162 input inputs are run twice, once with no load balancing iterations, and once with ten load balancing iterations. The best metric is reported and recorded. Three figures for each test cases are presented below: the first figure will show the metric behavior for no iterations, the second figure will show the metric behavior for each input run with ten load balancing iterations, and the third figure will show a ratio of the ten iteration runs over the no iteration runs.

Figure 8 shows the metric behavior for Fig. 5. The maximum metric value is 24.7650, and occurs when Fig. 5 is run with 8x8 subsets and a maximum triangle area of 1.6 cm^2. The minimum metric value is 1.0016 and occurs when Fig. 5 is run with 4x4 subsets and a maximum triangle area of 0.04 cm^2. The z axis in all figures is the value of the metric.

Figure 9 shows the metric behavior for Fig. 5 after 10 load balancing iterations. The maximum metric value is 5.0538 and occurs when Fig. 5 is run with 10x10 subsets and a maximum triangle area of 1.2 cm^2. The minimum metric value is 1.0017 and occurs when Fig. 5 is run with 4x4 subsets and a maximum triangle area of 0.04 cm^2.

Figure 10 shows the metric behavior for Fig. 6. The maximum metric is 22.6654 and occurs when Fig. 6 is run with 8x8 subsets with a maximum triangle area of 1.8 cm^2. The minimum metric is 1.0024 and occurs when Fig. 6 is run with 2x2 subsets with a maximum triangle are of 0.01 cm^2.

Figure 11 shows the metric behavior for Fig. 6 after ten

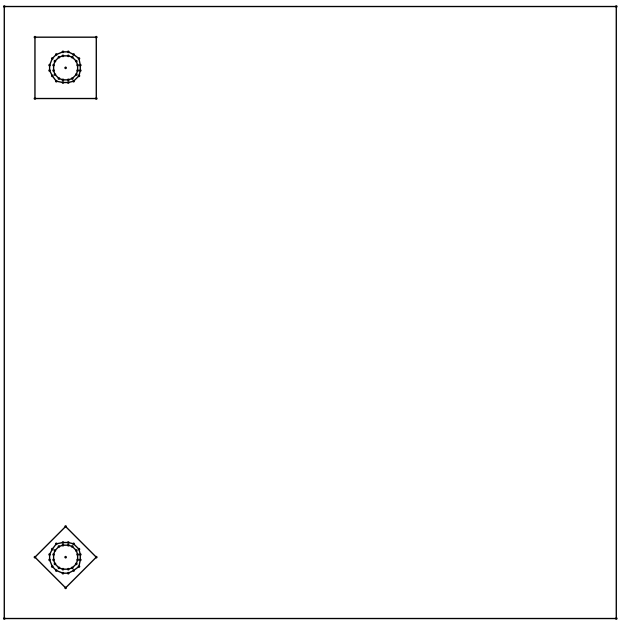

Fig. 6. The second test case used in order to test effectiveness and convergence of the load balancing metric.

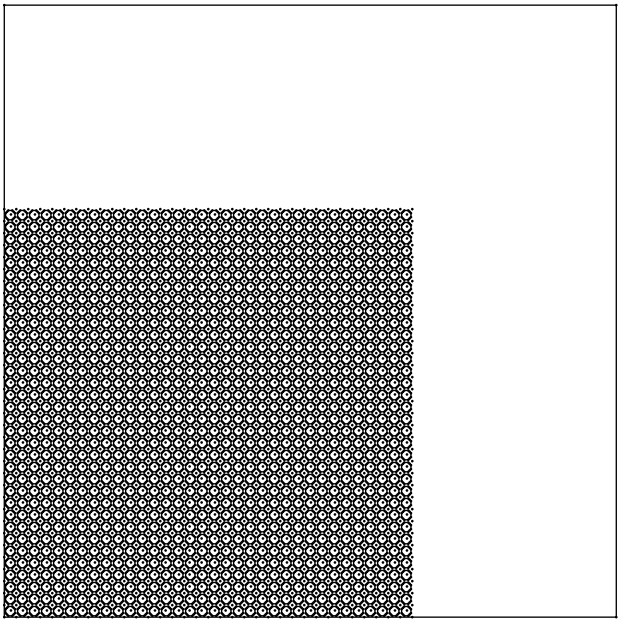

Fig. 7. The third test case used in order to test effectiveness and convergence of the load balancing metric.

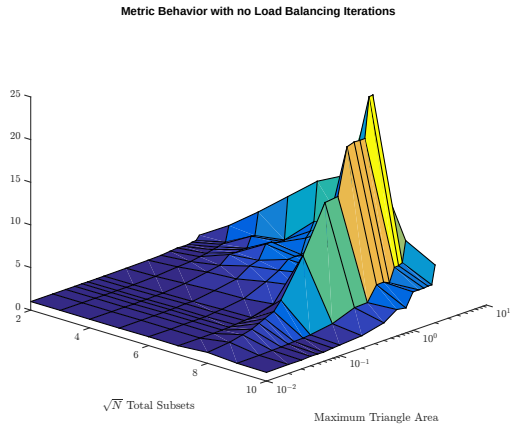

Fig. 8. The metric behavior of the first test case run with no load balancing iterations.

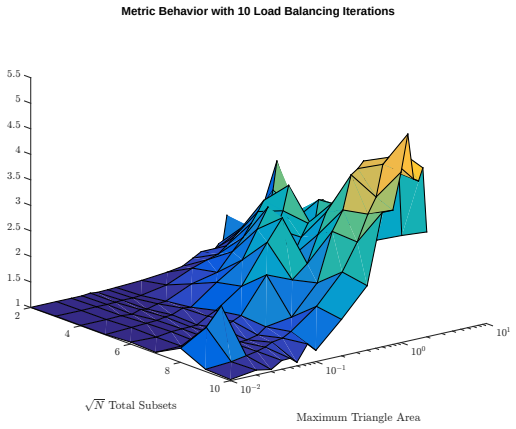

Fig. 9. The metric behavior of the first test case run with 10 load balancing iterations.

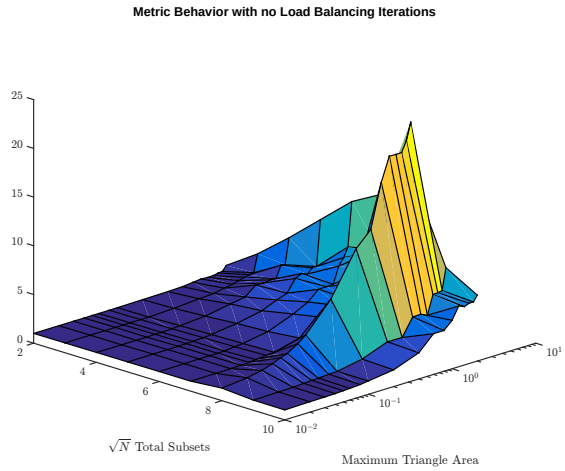

Fig. 10. The metric behavior of the second test case run with no load balancing iterations.

load balancing iterations. The maximum metric is 3.9929 and occurs when Fig. 6 is run with 10x10 subsets with a maximum triangle area of 1.8 cm^2. The minimum metric is 1.0024 and occurs when Fig. 6 is run with 2x2 subsets with a maximum triangle are of 0.01 cm^2.

Figure 12 shows the metric behavior for Fig. 7. The maximum metric is 2.6489 and occurs when Fig. 7 is run with 10x10 subsets with a maximum triangle area of 1.8 cm^2. The minimum metric is 1.0179 and occurs when Fig. 7 is run with 2x2 subsets with a maximum triangle are of 0.08 cm^2.

Figure 13 shows the metric behavior for Fig. 7 after ten load balancing iterations. The maximum metric is 2.2660 and occurs when Fig. 7 is run with 10x10 subsets with a maximum triangle area of 0.4 cm^2. The minimum metric is 1.0021 and occurs when Fig. 7 is run with 2x2 subsets with the Triangle's coarsest possible mesh.

Because Fig. 7 has more features and is more symmetric of a problem, the initial load balancing metric will not be as large as the load balancing metric of Figs. 6 and 5. As a result, the improvement in the load balancing metric after 10 iterations will not be as great in problems similar to Fig. 7.

Good improvement is seen throughout all three test cases for all three inputs, particularly the first two test cases, which were initially very unbalanced. However, there were many inputs run that had problems with $f > 1.1$, which means many problems were unbalanced by more than 10%. The user will not always have the luxury of choosing the number of subsets they want the problem run with, as this directly affects the number of processors the problem will be run with. Certain problems will require more processors and will require minimizing the total number of cells in the domain for the problem to complete running in a reasonable amount of time. As a result, improvements to the algorithm must be made.

This can be done by changing how the cut lines are redistributed. Instead of changing entire row and column widths,

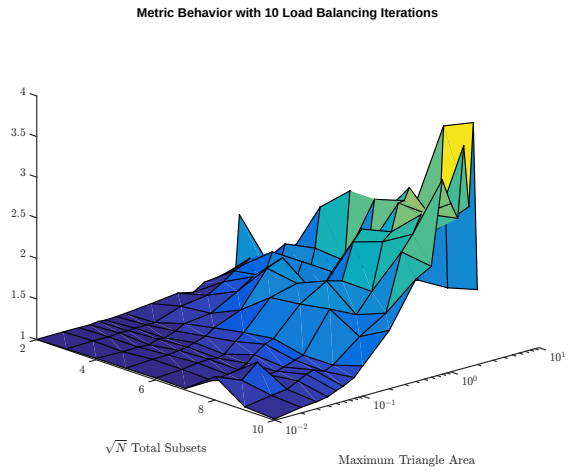

Fig. 11. The metric behavior of the second test case run with 10 load balancing iterations.

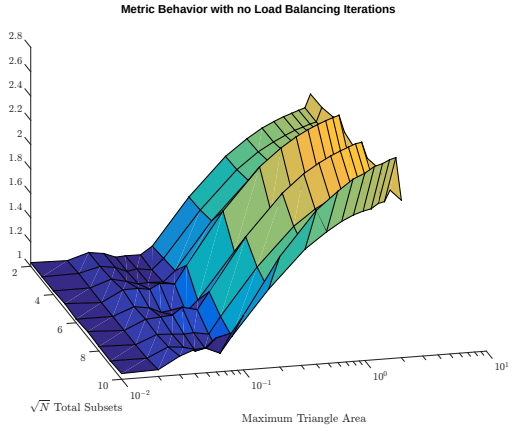

Fig. 12. The difference in metric behavior of the third test case with no load balancing iterations.

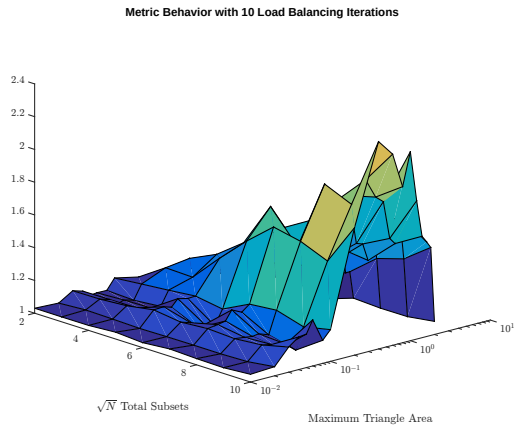

Fig. 13. The difference in metric behavior of the third test case after ten load balancing iterations.

the cut lines can be moved on the subset level. However, this can sacrifice the strict orthogonality that PDT currently utilizes to scale so well on a massively parallel scale[1]. Changes to the performance model and the scheduler would have to be made.

Another option is to implement domain overloading[1], which is the logical extension of the work presented in this paper. This would involve processors owning different numbers of subsets, with no restriction on these subsets being contiguous. This would be the most effective method at perfecting this algorithm, and would lead to less problems being unbalanced by more than 10%.

3. 2D and 2D Extruded Meshing Capability

To illustrate the newly implemented unstructured meshing capability in PDT, a neutron graphite experiment, modeled at Texas A&M University as part of the DOE's NNSA PSAAP program, is used as an example. Figure 15 shows the 2D mesh of the IM1 problem before and after 7 load-balancing iterations. The metric before any load balancing iterations is 42.15, and 2.99 after 7 iterations. The 3D extruded mesh is shown in Fig. 16.

V. CONCLUSIONS

In conclusion, the load balancing algorithm outlined in the Load Balancing Method section works well for more symmetric problems with a lot of features, and even works well for particularly unbalanced problems. As shown in the results, its effectiveness depends on the maximum triangle area used, and the number of subsets the user chooses to decompose the problem domain into.

Good improvement is seen throughout all three test cases for all three inputs, particularly the first two test cases, which were initially very unbalanced. However, there were many inputs run that had problems with $f > 1.1$, which means

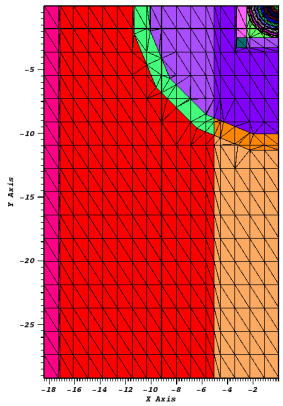

Fig. 14. The 2D mesh of the IM1 problem with no load balancing iterations.

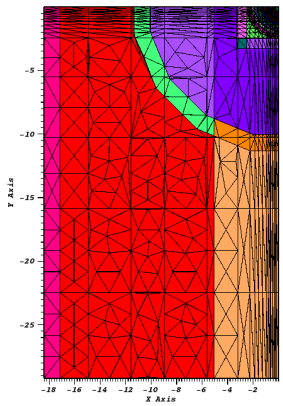

Fig. 15. The 2D mesh of the IM1 problem with 7 load balancing iterations.

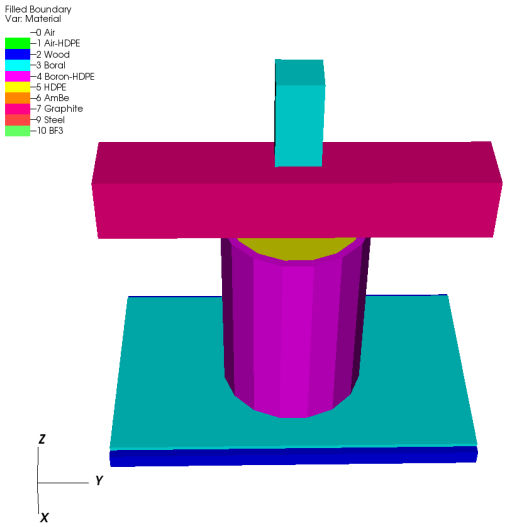

Fig. 16. The 3D extruded view of the IM1 problem (HDPE cylinder with AmBe source; graphite block; detector and detector shroud).

many problems were unbalanced by more than 10%. The user will not always have the luxury of choosing the number of subsets they want the problem run with, as this directly affects the number of processors the problem will be run with. Certain problems will require more processors and will require minimizing the total number of cells in the domain for the problem to complete running in a reasonable amount of time. As a result, improvements to the algorithm must be made.

This can be done by changing how the cut lines are redistributed. Instead of changing entire row and column widths, the cut lines can be moved on the subset level. However, this can sacrifice the strict orthogonality that PDT currently utilizes to scale so well on a massively parallel scale[1]. Changes to the performance model and the scheduler would have to be made. This method is currently being implemented.

Another option is to implement domain overloading[1], which is the logical extension of the work presented in this paper. This would involve processors owning different numbers of subsets, with no restriction on these subsets being contiguous. This would be the most effective method at perfecting this algorithm, and would lead to less problems being unbalanced by more than 10%.

VI. ACKNOWLEDGMENTS

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

REFERENCES

1. M. A. ET AL, “Provably Optimal Parallel Transport Sweeps with Non-Contiguous Partitions,” in “ANS MC2015-Joint International Conference on Mathematics and Computation (M&C), Supercomputing in Nuclear Applications (SNA) and the Monte Carlo (MC) Method,”

(2015).

2. J. C. RAGUSA, "Discontinuous finite element solution of the radiation diffusion equation on arbitrary polygonal meshes and locally adapted quadrilateral grids," *Journal of Computational Physics*, **280**, 195–213 (2015).
3. T. S. BAILEY, M. L. ADAMS, B. YANG, and M. R. ZIKA, "A Piecewise Linear Discontinuous Finite Element Spatial Discretization of the S_N Transport Equation for Polyhedral Grids in 3D Cartesian Geometry," *Journal of Computational Physics*, **227**, 3738–3757 (2008).
4. A. T. TILL, *Finite Elements with Discontiguous Support for Energy Discretization in Particle Transport*, Ph.D. thesis, Texas A&M University, College Station, TX (2015).
5. J. SHEWCHUK, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," in M. C. LIN and D. MANOCHA, editors, "Applied Computational Geometry: Towards Geometric Engineering," Springer-Verlag, *Lecture Notes in Computer Science*, vol. 1148, pp. 203–222 (May 1996), from the First ACM Workshop on Applied Computational Geometry.
6. M. A. ET AL, "Provably Optimal Parallel Transport Sweeps on Regular Grids," in "International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering," (2013).
7. T. E. ET AL., "Denovo: A New Three-Dimensional Parallel Discrete Ordinates Code in Scale," *Nuclear Technology*, **171**, 171–200 (2010).
8. R. ALCOUFFE, R. BAKER, S. TURNER, and J. DAHL, "PARTISN manual," Tech. rep., LA-UR-02-5633, Los Alamos National Laboratory (2002).