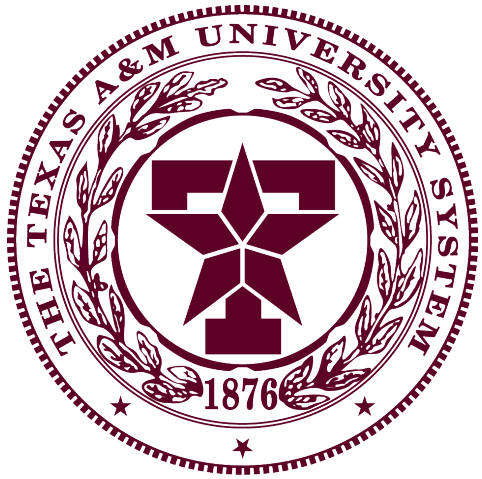


Load Balancing Unstructured Meshes for Massively Parallel Transport Sweeps

Tarek Ghaddar
Chair: Dr. Jean Ragusa
Committee: Dr. Jim Morel, Dr. Bojan Popov

Texas A&M University

Motivation

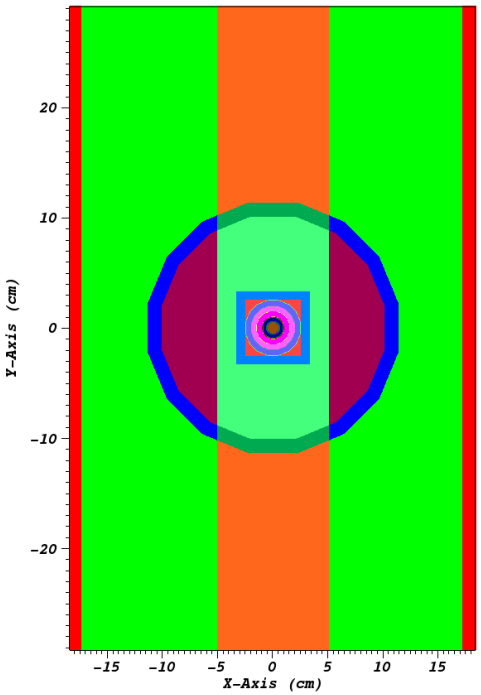
- When running any massively parallel code, load balancing is a priority in order to achieve the best possible parallel efficiency.
- A load balanced problem has an equal number of degrees of freedom per processor.
- Load balancing a logically Cartesian mesh is not difficult, as the user specifies the number of cells being used.
- In an unstructured mesh, the user cannot always specify the number of cells they want per processor, and obtaining a load balanced problem is more difficult.

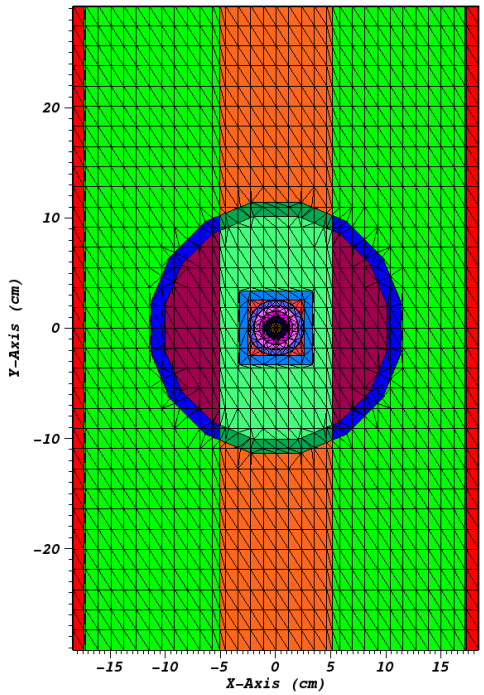
PDT

- All work presented in this thesis was implemented in Texas A&M's massively parallel deterministic transport code, PDT.
- It is capable of multi-group simulations and employs discrete ordinates for angular discretization.
- Features steady-state, time-dependent, criticality, and depletion simulations. It 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.

The Triangle Mesh Generator

- Unstructured meshes in PDT are generated using the Triangle Mesh Generator.





The Transport Equation

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

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

The Transport Equation

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

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

$$\text{for } 1 \leq g \leq G$$

The Transport Equation

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

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

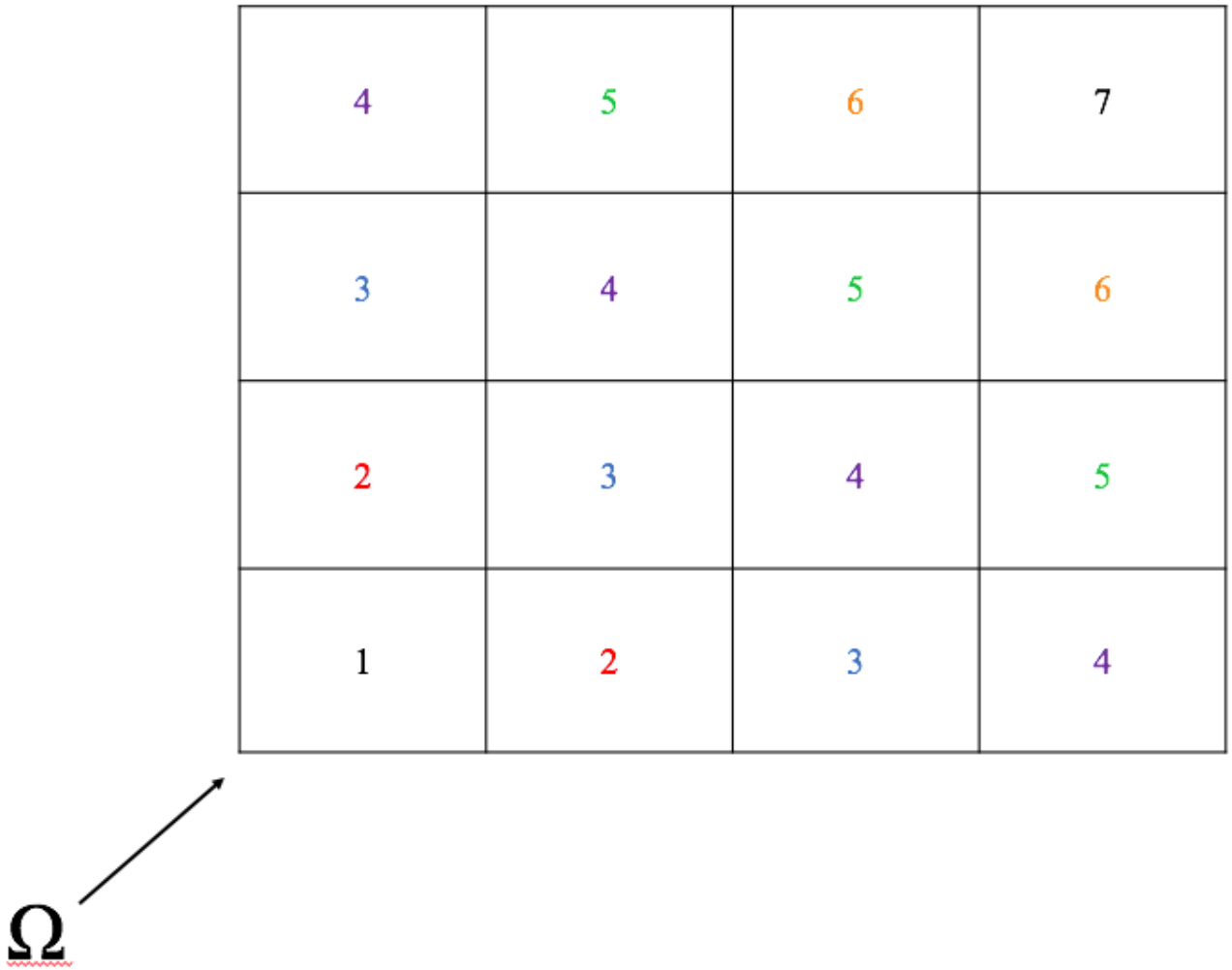
$$\vec{\Omega}_m \cdot \vec{\nabla}\psi_m^{(l+1)}(\vec{r}) + \Sigma_t \psi_m^{(l+1)}(\vec{r}) = q_m^{(l)}(\vec{r})$$

The Transport Sweep

A parallel sweep algorithm is defined by three properties:

- 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 Sweep



Aggregation

- $A_x = \frac{N_x}{P_x}$, 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
- $N_g = \frac{G}{A_g}$
- $N_m = \frac{M}{A_m}$
- $N_k = \frac{N_z}{P_z A_z}$
- $N_k A_x A_y A_z = \frac{N_x N_y N_z}{P_x P_y P_z}$

Parallel Efficiency

$$\epsilon = \frac{T_{\text{task}} N_{\text{tasks}}}{[N_{\text{stages}}][T_{\text{task}} + T_{\text{comm}}]}$$
$$= \frac{1}{[1 + \frac{N_{\text{idle}}}{N_{\text{tasks}}}][1 + \frac{T_{\text{comm}}}{T_{\text{task}}}]}$$

$$T_{\text{comm}} = M_L T_{\text{latency}} + T_{\text{byte}} N_{\text{bytes}}$$
$$T_{\text{task}} = A_x A_y A_z A_m A_g T_{\text{grind}}$$

Metric Definitions

- $f = \frac{\max\limits_{ij}(N_{ij})}{\frac{N_{tot}}{I \cdot J}}$
- $f_I = \max\limits_{i}[\sum_j N_{ij}] / \frac{N_{tot}}{I}$
- $f_J = \max\limits_{j}[\sum_i N_{ij}] / \frac{N_{tot}}{J}$

Algorithm

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

Redistribution

```
stapl::array_view num_tri_view
stapl::array_view offset_view
//offset_view stores partial sum of num_tri_view
stapl::partial_sum(num_tri_view)
//We now have a cumulative distribution stored in offset_view
for (i = 1:X.size()-1)
{
    vector <double> pt1 = [CutLines(i-1), offset_view(i-1)]
    vector <double> pt2 = [CutLines(i), offset_view(i)]
    ideal_value = i*(N_tot/num_subsets_X);
    x_val = X-intersect(pt1,pt2,ideal_value);
    if ((x_val > x_cuts[i-1] && x_val< x_cuts[i])
      || equal(x_val,x_cuts[i]) || equal(x_val,x_cuts[i-1])
    {
      X[i] = x_val;
    }
}
```