Load Balancing by Dimension

Introduction

As has been proven over the last few months, we need an improved load balancing algorithm. The most popular option that can be implemented in PDT now is the now-called Load Balancing by Dimension (LBD) algorithm. The gist of the algorithm is to first move X cut planes and use the column wise load balance metric, f_I, to determine when to stop redistribution, and then to load balance each row WITHIN each column.

This will of course lead to a major issue: the determination of each subset's neighbor is no longer as easy. Previously, we were able to use a simple i, j indexing for each subset, which makes the determination of neighbors trivial. Now, with cut planes in Y no longer lining up perfectly, a subset can have mutiple neighbors to the right or left, not just one of each. Figure 1 shows the i, j functionality, while Fig. 3 shows the current predicament.

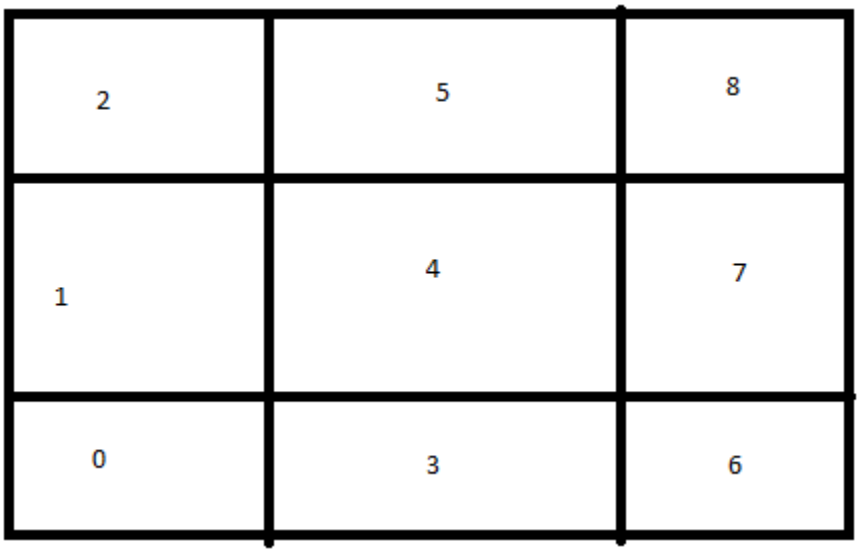

Figure 1. The previous load balancing algorithm result that allows for the i, j system of determining the four neighbors for each subset. Numbers are subset IDs.

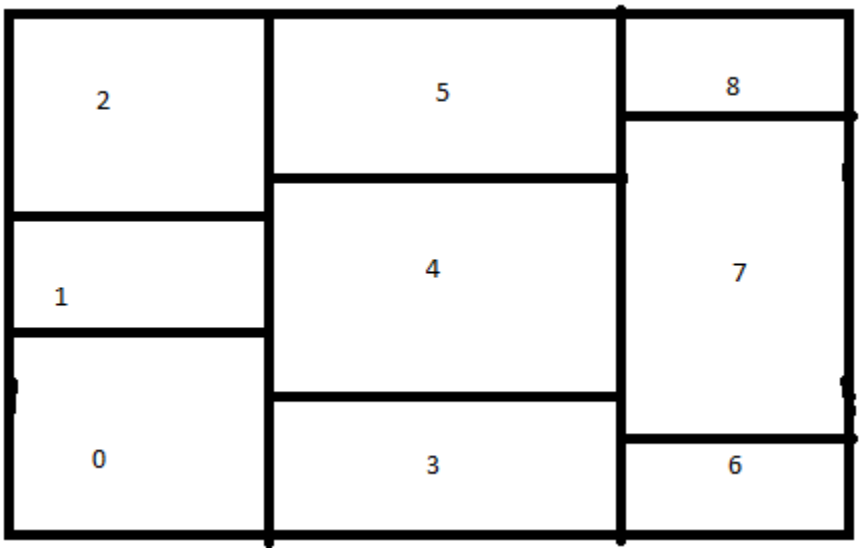

Figure 2. A load balancing by dimension example result that disallows the use of the i, j system of determining neighbors. Numbers are subset IDs.

Pseudocode

```
1  //This is pseudocode for the load balancing by dimension algorithm.
   function LOAD_BALANCE_BY_DIMENSION()
   {
     //All subsets must be meshed initially.
5    MESH all subsets;

     //Caluclate the column-wise load balance metric, f_I
     CALCULATE f_I;
     //Calculate the row-wise load balance metric
10   CALCULATE f_J;

     if (f_I <= f_J)
     {
       while (f_I > column_tolerance)
15     {
         //If the column wise metric is still greater than the tolerance,
         Redistribute(x_cuts);
         //Remesh
         MESH all subsets;
20     }
     }
     else
     {
       while(f_J > row_tolerance)
25     {
         //If the row-wise metric is still greater than the tolerance.
         Redistribute(y_cuts);
         //Remesh
         MESH all subsets;
30     }
     }

     //Once we have columns (rows) balanced, we have to load balance the rows(columns) in
         each column(row) now.
     for each column(row)
35   {
       //Calculate the row-wise(column-wise) metric for each column(row)
       CALCULATE f_I(f_J);
       while (f_I(f_J) > tolerance)
       {
40       //Redistribute the rows in this column.
         Redistribute(y_cuts_column(x_cuts_row));
         //All subsets must be meshed initially.
         MESH all subsets;
       }
45   }
   }

   //Psuedocode to build and store and undirected task dependence graph.
   function BUILD_UNDIRECTED_TDG(x_cuts,y_cuts)
50 {
     //We change how the y_cuts(x_cuts) are stored. They are now stored BY COLUMN(ROW).
         We need to store them this way in order to properly build our mesh, and properly
         define subset boundaries.
     //Each subset knows it's own xmin/ymin, xmax/ymax boundaries. We can base this on a
         loose i,j framework.
```

```
    //We know that each subset will have a neighbor above and below it (except if it's
        on a global y boundary). These neighbors never change and we do not need to
        account for them.
    //The only neighbors in flux after a load balancing iteration are the subsets right
        and/or left neighbors.

    //We should be able to replace column with row in the case that the rows have cuts
        going across the entirety of the domain.

    //If the current subset is in the first column, we only need to look to our right
        neighbor column.
    if (column == first_column)
    {
      //Pull the y_cuts for the second column.
      second_column_y_cuts = y_cuts[second_column];
      //Loop over all the y_cuts in this column.
      for (i = 0; i < num_rows-1; ++i)
      {
        //Check if this y_cut is less than or equal to the maximum y boundary of this
            subset.
        if ( second_column_y_cuts[i] < y_max )
        {
          //Check if this y_cut is greater than or equal to the minimum y boundary of
              this subset.
          if ( second_column_y_cuts[i] > y_min )
          {
            //Register both subsets that share that y_cut as neighbors.
          }
          else if ( second_column_y_cuts[i] == y_min )
          {
            //Register the subset that has this y_cut as its ymin as a neighbor.
          }
        }
        else if ( second_column_y_cuts[i] == y_max)
        {
          //Register the subset that has this y_cut as its ymax as a neighbor.
        }
        //This y_cut is greater than the maximum y boundary of this subset.
        else
        {
          //Check if we're looking at the first cut line in the column.
          if ( i == 0 )
          {
            //Register the subset that has that y_cut as its ymax as a neighbor.
          }
          else if(second_column_y_cuts[i-1] < y_max)
          {
            //Register the subset that has second_colum_y_cut[i] as its ymax as a
                neighbor.
          }
        }
      }
    }
    //If the subset is in the last column, we only need to look to our left neighbor
        column.
    else if(column == last_column)
    {
      //Pull the y_cuts for the second to last column.
      left_column_y_cuts = y_cuts[last_column-1];
```

```
      //Loop over all the y_cuts in this column.
      for (i = 0; i < num_rows-1; ++i)
105   {
        //Check if this y_cut is less than or equal to the maximum y boundary of this
            subset.
        if ( left_column_y_cuts[i] < y_max )
        {
          //Check if this y_cut is greater than or equal to the minimum y boundary of
              this subset.
110       if ( left_column_y_cuts[i] > y_min )
          {
            //Register both subsets that share that y_cut as neighbors.
          }
          else if ( left_column_y_cuts[i] == y_min )
115       {
            //Register the subset that has this y_cut as its ymin as a neighbor.
          }
        }
        else if ( left_column_y_cuts[i] == y_max)
120     {
          //Register the subset that has this y_cut as its ymax as a neighbor.
        }
        //This y_cut is greater than or equal to the maximum y boundary of this subset.
        else
125     {
          //Check if we're looking at the first cut line in the column.
          if ( i == 0 )
          {
            //Register the subset that has that y_cut as its ymax as a neighbor.
130       }
          else if(left_column_y_cuts[i-1] < y_max)
          {
            //Register the subset that has second_colum_y_cut[i] as its ymax as a
                neighbor.
          }
135     }
      }
    }
    //If the current subset is located in an interior column, we need to check both the
        right and left neighbor columns.
    else
140 {
      left_column_y_cuts = y_cuts[current_column - 1];
      right_column_y_cuts = y_cuts[current_column + 1];
      //Loop over all the y_cuts in this column.
      for (i = 0; i < num_rows-1; ++i)
145   {
        //Check if this y_cut is less than or equal to the maximum y boundary of this
            subset.
        if ( left_column_y_cuts[i] < y_max )
        {
          //Check if this y_cut is greater than or equal to the minimum y boundary of
              this subset.
150       if ( left_column_y_cuts[i] > y_min )
          {
            //Register both subsets that share that y_cut as neighbors.
          }
          else if ( left_column_y_cuts[i] == y_min )
155       {
```

```
            //Register the subset that has this y_cut as its ymin as a neighbor.
          }
        }
        else if ( left_column_y_cuts[i] == y_max)
        {
          //Register the subset that has this y_cut as its ymax as a neighbor.
        }
        //This y_cut is greater than the maximum y boundary of this subset.
        else
        {
          //Check if we're looking at the first cut line in the column.
          if ( i == 0 )
          {
            //Register the subset that has that y_cut as its ymax as a neighbor.
          }
          else if(left_column_y_cuts[i-1] < y_max)
          {
            //Register the subset that has second_colum_y_cut[i] as its ymax as a
                neighbor.
          }
        }
        //Check if this y_cut is less than or equal to the maximum y boundary of this
            subset.
        if ( right_column_y_cuts[i] < y_max )
        {
          //Check if this y_cut is greater than or equal to the minimum y boundary of
              this subset.
          if ( right_column_y_cuts[i] > y_min )
          {
            //Register both subsets that share that y_cut as neighbors.
          }
          else if ( right_column_y_cuts[i] == y_min )
          {
            //Register the subset that has this y_cut as its ymin as a neighbor.
          }
        }
        else if ( right_column_y_cuts[i] == y_max)
        {
          //Register the subset that has this y_cut as its ymax as a neighbor.
        }
        //This y_cut is greater the maximum y boundary of this subset.
        else
        {
          //Check if we're looking at the first cut line in the column.
          if ( i == 0 )
          {
            //Register the subset that has that y_cut as its ymax as a neighbor.
          }
          else if( right_column_y_cuts[i-1] < y_max )
          {
            //Register the subset that has second_colum_y_cut[i] as its ymax as a
                neighbor.
          }
        }
      }
    }
    //Each subset cleans up it's duplicates and we now have a map of the neighbors for
        each subset!
  }
```

Examples

Let's look at Fig. 3 once more.

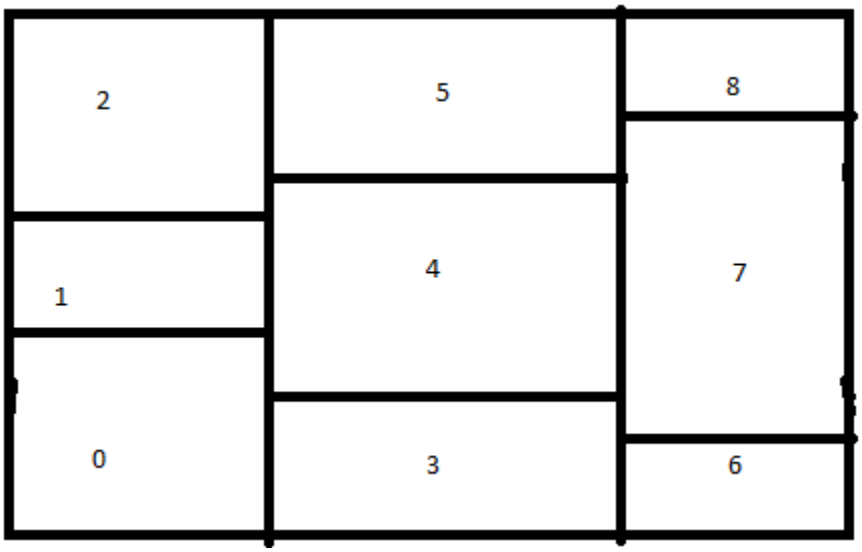

Figure 3. A load balancing by dimension example result that disallows the use of the i, j system of determining neighbors. Numbers are subset IDs.

Let's take a look at subset 0 and follow through the pseudocode in the BUILD_UNDIRECTED_GRAPH function (line 42) :

- Subset 0 is located in the first column, so it's only potential neighbors are to the right in the second column.
- We pull the cut line information for the second column, and we look at the two cut interior cut lines, not the cut lines on the global boundary.
- Looking at the first cut line in the second column, we see that it is less than subset 0's y-max, and greater than subset 0's y-min. Therefore, the two subsets that share this cut line as a boundary, subsets 3 and 4, are added as neighbors to subset 0.
- Looking at the second cut line in the second column, we see that it is above subset 0's y-max, so we check that the previous cut line is below subset 0's ymax. It is, so subset 4 is added as a neighbor again.
- Duplicates are removed, and subsets 3 and 4 are subset 0's neighbors to the right/left.

We take a look at subset 4 and follow through the pseudocode in the same way:

- Subset 4 is not located in the first or last column, so it's potential neighbors are in the columns to the left and right.

- We pull the cut line information for the right and left columns, and we look at the two cut interior cut lines, not the cut lines on the global boundary.

- Looking at the first cut line in the left column, we see that it is less than subset 4's ymax, and greater than subset 4's ymin. Therefore, the two subsets that share this cut line as a boundary, subsets 0 and 1, are added as neighbors to subset 4.

- Looking at the second cut line in the left column, we see that it is less than subset 4's ymax, and greater than subset 4's ymin. Therefore, the two subsets that share this cut line as a boundary, subsets 1 and 2, are added as neighbors to subset 4.

- Looking at the first cut line in the right column, we see that it is less than subset 4's ymax, and less than subset 4's ymin. No neighbor is added.

- Looking at the second cut line in the right column, we see that it is greater than subset 4's ymax. The previous cut line is less than subset 4's ymax, so the subset that has the second cut line as its ymax, subset 7, is added as a neighbor to subset 4.

- Duplicates are removed, and subsets 0,1,2, and 7 are subset 4's neighbors to the right/left.