EFFICIENT KD-TREE BASED MESH REDISTRIBUTION FOR DATA REMAPPING ALGORITHMS

Navamita Ray [1*], Daniel Shevitz [1], Yipeng Li [2†], Rao Garimella [3], Angela Herring [4], Evgeny Kikinzon [1], Konstantin Lipnikov [3], Hoby Rakotoarivelo [3], Jan Velechovsky [5]

[1] *Computer, Computational and Statistical Sciences, CCS-7, Los Alamos National Laboratory, Los Alamos, NM, USA*
[2] *Department of Applied Mathematics and Statistics, StonyBrook University*
[3] *Theoretical Division, T-5, Los Alamos National Laboratory, Los Alamos, NM, USA*
[4] *X-Computational Physics, XCP-4, Los Alamos National Laboratory, Los Alamos, NM, USA*
[5] *X-Computational Physics, XCP-2, Los Alamos National Laboratory, Los Alamos, NM, USA*

ABSTRACT

In this paper, we present a new mesh redistribution algorithm developed for the parallel data remapping library Portage. During distributed memory parallel remapping, source and target meshes are partitioned independently of each other, requiring a mesh redistribution so that all cells on the target mesh partition are covered by source mesh partition cells. Our new algorithm uses a KD-tree data structure to capture the general shape of the target mesh and find an improved overlap between source and target mesh partitions so as to redistribute fewer cells. We present numerical results showing that the KD-tree method reduces memory storage requirements for the redistributed mesh on each partition and is faster than the old bounding box method.

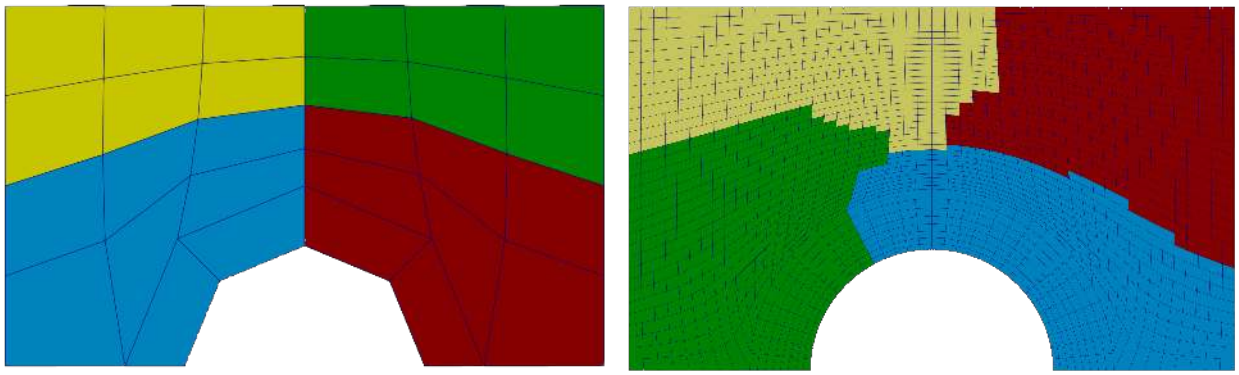
Keywords: mesh redistribution, data remapping, parallel algorithms

1. INTRODUCTION

Data remapping is used in many multi-physics applications to transfer numerical fields from a source mesh to target mesh. For example, in Arbitrary Lagrangian-Eulerian (ALE) methods ([1], [2], [3], [4]) for hydrodynamics applications, the Lagrangian mesh is moved along with the fluid flow for some time steps before the cells distort excessively. Then, the mesh nodes are rezoned or smoothed to yield a better quality mesh, and finally, the fields on the Lagrangian source mesh are interpolated to the improved target mesh. In other multi-physics applications ([5], [6], [7]), where different physics domains depend on each other through shared domain boundaries, there is a need to transfer fields along the domain boundary to solve the governing equations of that component.

In order to remap data in parallel onto the target mesh, the target mesh partition on any Message Passing Interface (MPI) rank should have all the source mesh cells covering it available on the same rank. This is a requirement of many remap methods, particularly, of conservative field remap methods, where quantities like intersection volumes, field gradients, etc., are needed for data interpolation. For parallel remapping on distributed systems, generally the source and target meshes are partitioned independently of each other. This can lead to scenarios where the target mesh partition is only partially (or not at all) covered by the source mesh partition on the same MPI rank. For example, Figure 1 shows a partitioning of a simple source and target mesh on four MPI ranks where the partitions are color-coded, so that source and target mesh partitions on the same rank have the same color. The

*Corresponding author
†Current affiliation OneFlow, China



(a) Source Mesh Partitions

(b) Target Mesh Partitions

Figure 1: Example source and target mesh partitions on four ranks. The colors correspond to the rank of the mesh partition.

yellow source mesh partition only covers part of the yellow target mesh partition, whereas the green target mesh partition is not covered at all by the green source mesh partition. To perform the remapping correctly, we must perform a mesh redistribution, i.e. bring the necessary source mesh information from all other MPI ranks to each target rank.

Portage [8] is a numerical library, which provides a suite of numerical algorithms for remapping fields from a source mesh to a target mesh. Currently Portage uses a coarse-grained bounding box based overlap detection algorithm to redistribute the meshes. While this method is failsafe, it also frequently sends unnecessary source data and with increased execution time and memory usage.

To detect if the target mesh partitions on other ranks overlap with the source mesh partition on the current rank, we need to have sufficient information about the shape of the target mesh partitions. We also need to figure out how much the source mesh partition on the current rank overlaps with the target mesh partitions on other ranks. Ideally, the redistribution process should send only as much information as necessary to other partitions. In this paper, we present a new method to perform better overlap detection and more precisely control the information copied across ranks.

In [9], two mesh redistribution algorithms are described suitable for distributed systems. They use a rendezvous technique wherein a third decomposition is computed so that both the source and target mesh partitions overlap completely on this third decomposition. The recursive coordinate bisectioning (RCB) partitioning strategy is used to obtain this third decomposition. The decomposition is primarily for nodal remap, where one needs to know the source cell containing each target node, so that the nodal values from the source cell are interpolated at the target node. The Data Transfer Kit ([10]) a software library designed to provide parallel services for mesh and geometry searching and data transfer. The algorithms implemented in Data Transfer Kit are based on the rendezvoud algorithms described in [9]. In [11], a dynamically load-balancing algorithm for parallel particle redistribution using KD-trees for particle tracing applications is described. In the algorithm, each process starts with a statically partitioned axis-aligned data block that partially overlaps with neighboring blocks in other processes along with a dynamically determined k-d tree leaf node that bounds the active particles for computation. The particles are periodically redistributed based on a constrained KD-tree decomposition, which is limited to the expanded overlapping layers.

Our method differs from these approaches on multiple aspects. Portage is more general, and supports both nodal and cell-value remapping algorithms as well as other remapping algorithms. The mesh redistribution needs to satisfy the conditions of all such remapping algorithms. Also, computing a new partitioning of the source and target meshes might be computationally expensive as the library might be used as part of a multi-physics application where a remap needs to happen every time step of the simulation.

The K-dimensional tree (KD-tree, [12]) is a data structure that splits K-dimensional data for efficient range queries and K-neighbor queries. Our method uses the KD-tree data structure to capture the general shape of the target mesh partition, which is subsequently used to detect the specific source cells that intersect with this target mesh partition shape approximation. Based on this refined overlap detection, we send only part of the mesh from an overlapping source mesh partition to the target mesh partition rank. We control the amount of information copied (sent) across parti-

tions by controlling the depth of the KD-tree on the target mesh partitions. We performed numerical studies to show the improvements in both memory and time by the new method in comparison to the current approach. In section 2, we start with a brief overview of the default bounding box algorithm. Section 3 describes the new approach to mesh redistribution. In 4, we present numerical studies comparing the new algorithm with the bounding box method.

2. BOUNDING BOX ALGORITHM

The coarsest geometric representation of a general shape is its axis aligned bounding box. The bounding box algorithm implemented in Portage utilizes this description, and is a simple rendezvous algorithm. Bounding boxes of both the target and source mesh partitions are used to detect overlaps. The key steps in the algorithm are as follows:

1. On each rank, the axis aligned bounding box of the target and source mesh partitions are constructed.
2. Each rank broadcasts the bounding box description of its target mesh partition to all ranks, so that each rank has an approximated shape of the global target mesh.
3. On each rank, if the source bounding box intersects with any received target bounding box, then all the cells in the source mesh partition are sent to the target rank.

By design, this method is conservative in its approach. As a result, it almost always overestimates the number of cells that must be copied over to the target partitions. For example, even when the source mesh partition bounding box is only slightly intersecting any of the received target mesh partition bounding boxes, the overlap detection deduces that they intersect, and sends the whole source mesh partition to the target rank. Due to this conservative overlap detection, it can happen that multiple source mesh partitions are migrated to a target rank which can lead to scenarios where almost the whole global source mesh is on a target rank after redistribution, resulting in significant increase in memory usage. In worst cases, the remap code can fail at runtime due to the large memory overload.

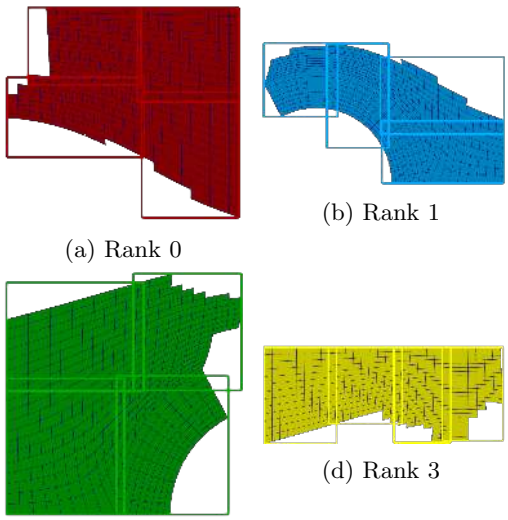
3. MESH REDISTRIBUTION USING A KD-TREE

In the new approach, we focus on improving all components of the overlap detection process. First, we use a KD-tree data structure to generate a better and controllable description of the target mesh shape. The representation is tunable, ranging from the coarsest one bounding box covering the target to the finest depth with a bounding box for each target cell. Second, we use an efficient search on the source mesh partition, again using a KD-tree data structure to obtain the list of candidate cells that intersect with the target bounding boxes. Finally, we migrate only part of the mesh from an overlapping source mesh partition to the target mesh partition rank.

In this section, we describe the key steps (listed below) of the new algorithm in more detail.

1. **Target Mesh Shape Approximation**: On each rank, generate an approximation of the target mesh partition shape using a KD-tree data structure and broadcast the approximate target shape to all ranks. The approximated target mesh partition shape is a list of target bounding boxes depending on the depth of the KD-tree representation.
2. **Overlap Detection**: On each rank, find the cells in the source mesh partition that overlap the target mesh partition shapes received from other ranks. This step results in obtaining lists of source cells, one list of candidate source cells for each target rank it detects an overlap with.
3. **Mesh Migration**: Each rank sends the overlapping source cells along with their field data to the target ranks.

Figures 1, 2 and 3 show an example of the overlap detection process. Figures 1a and 1b are a source and target mesh partitioned into four ranks where the source and target parts on the same rank have the same color. The bounding boxes of a depth 2 KD-tree over the target mesh partitions are shown in Figures 2a, 2b, 2c and 2d. Note that since a KD-tree is a binary tree at any fixed depth there will be a power of two number of bounding boxes ignoring incomplete filling due to an unbalanced tree. In Figure 3a, the aggregation of target bounding boxes from all target ranks on each source rank are shown. During the overlap detection, these bounding boxes are used to find lists of source cells that need to be sent to target partitions. For example, in Figure 3b, the source mesh partitions on green, blue and red ranks will detect the cells intersecting with the target bounding boxes from the yellow rank and select only the subset of the meshes that overlap these boxes.



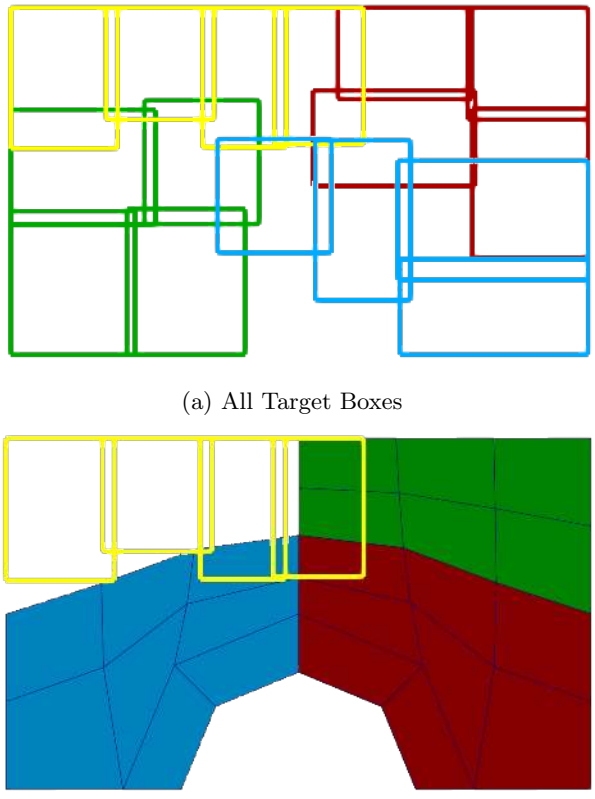
(a) Rank 0

(b) Rank 1

(c) Rank 2

(d) Rank 3

Figure 2: Target boxes constructed using a KD-tree at depth 2. Note the 4 bounding boxes per partition.



(a) All Target Boxes

(b) Source Target Overlap

Figure 3: The target box description of the target mesh globally as well as overlap of target bounding boxes on rank 3 with other source mesh partitions.

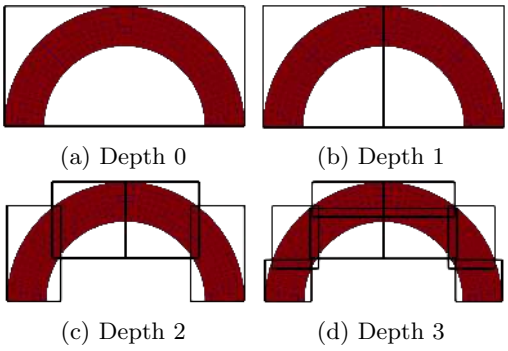
3.1 Target Mesh Shape Approximation

K-Dimensional tree(KD-tree) is a well-known space-partitioning data structure for organizing points in a k-dimensional space and is used for efficient searching. We use KD-trees for two purposes. First, we use the data structure to create a finer approximation of the target geometry as a collection of bounding boxes. Second, we use it to perform efficient searches for overlap detection between the target shape approximation and the source mesh partition as is described in section 3.2.

The KD-tree construction is agnostic to which mesh it is created on, so the description of the KD-tree construction uses the term mesh instead of target mesh. Indeed, we compute KD-trees on both the source and target mesh partitions, albeit for different purposes. In our KD-tree construction, each node at any depth is a bounding box. We begin by computing the axis aligned bounding box of each cell by using the minimum and maximum coordinates in each dimension. For each such box, we next compute its centroid. The construction algorithm takes as input the set of bounding boxes, the point set consisting of bounding box centroids, and the depth up to which the tree is to be constructed. The space partitioning uses the point set whereas the bounding boxes are used to construct the nodes of the tree.

At any depth in the tree, the parent node is the encapsulating bounding box of a set of cells. We next find which axis or direction (x/y/z) should be used to partition the point space (consisting of the bounding box centroids). We choose the axis with the longest side of the bounding box of the current node as the cutting direction. Once a direction has been chosen, we group the cells under the node into a left and a right set, where the left set has cells with centroid values less than the median along the cutting direction. The left and right children are now constructed out of the cells in the left and right sets. The tree construction is either stopped at the depth provided, or continues until the full depth of the tree possible for the input set.

In Algorithm 1, we present the pseudocode for the KD-tree construction. The algorithm uses a stack data structure to construct nodes of the tree, where the root node is the bounding box of the entire mesh. We also maintain a permutation array of the cell ids which is used to store the partitioning of the space as the tree construction progresses. We begin by finding the longest side of the root node bounding box. The axis corresponding to the longest side is chosen as the cutting direction. We next permute the cell ids (as stored in the permutation array), so that the median of the array is the cell id with median coordinate value of the centroid corresponding to the cutting direction. This



(a) Depth 0 (b) Depth 1

(c) Depth 2 (d) Depth 3

Figure 4: The KD-tree based representation with increasing depths.

groups the list of cell ids into a left and right part, where the left part has cell ids with centroid coordinate less than the median along the cutting direction. Similarly, the right part constitutes of cell ids with centroid coordinate greater than and equal to the median along the cutting direction. The left and right child node bounding boxes are now computed by gathering all the bounding boxes of the cells making up that child. For each child, the pointers to the minimum and maximum of the permutation array are stored, so that for the next level, the median is found only for that part. These steps are followed until the desired of the tree is obtained or the full tree is constructed. Figures 4a, 4b, 4c and 4d show four depths of KD-tree based shape approximation. As we can see, with increasing depths the tree leaf bounding boxes capture better the shape of the mesh which leads to better overlap detection.

Using the above process, we construct a KD-tree over the target mesh partition. We construct the tree over the owned cells of the target mesh partition as there may be ghost layers during initial partitioning, which are owned by other ranks. The output of the construction is a list of leaf bounding boxes at a fixed depth in the tree. This list is then broadcast across all ranks, at the end of which each rank has an approximate description of the global target shape.

3.2 Overlap Detection

After the broadcast of the bounding boxes of the target mesh partition across ranks, each rank now has an approximate description of the whole target mesh shape in the form of bounding boxes and the target partitions they belong to. We next want to detect the list of source cells that intersect the target bounding boxes from a received rank. Instead of nested linear loops over all received target boxes, and over the source cells to detect which cells in the source intersect target bounding boxes, we use another KD-tree,

Algorithm 1 KD-tree Construction

Input: B: N bounding boxes
Input: C: N centroids of the bounding boxes
Input: L: depth of the tree
Output: leaves: Leaf bounding boxes

```
P ← Permutation array of size N
root ← Bounding box encompassing all input boxes
if L = 0 then return root
else
    stack: array storing tree node ids
    min_idx: array storing minimum index into leaves array for a node
    max_idx: array storing maximum index into leaves array for a node
    leaves: array storing leaf boxes
    current_depth = 0
    top ← 0
    stack[top] ← 0
    nextp ← 1
    min_idx[top] ← 0
    max_idx[top] ← N
    while top ≥ 0 do
        current_depth = current_depth + 1
        min = min_idx[top]
        max = max_idx[top]
        top--
        cut_dir = cutting direction based the longest side of the node
        mid = (min + max)/2
        Reorder part of P such that ∀i : C[i][cur_dir] ≤ C[mid][cut_dir]
        if mid = min ||current_depth= L then
            box = construct bounding box encompassing using boxes from min to mid
            Add box to leaves
        else
            box = construct bounding box encompassing using boxes from min to mid
            Add box to leaves
            top++
            stack[top] = nextp
            min_idx[top] = min
            max_idx[top] = mid
            nextp++
        end if
        if mid+1 = max ||current_depthL= L then
            box = construct bounding box encompassing using boxes from mid+1 to max
            Add box to leaves
        else
            box = construct bounding box encompassing using boxes from mid+1 to max
            Add box to leaves
            top++
            stack[top] = nextp
            min_idx[top] = mid+1
            max_idx[top] = max
            nextp++
        end if
    end while
end if
```

this time for efficient searching. We create the full tree on the source mesh partition (so that the leaf nodes are the bounding boxes of the source cells), and perform the search between the source tree and target bounding boxes. Since the average cost of look up is O(LogN), we can avoid a linear search over source cells. The pseudocode for the overlap detection algorithm is shown in Algorithm 2.

After the search, we end up with candidate lists of source cells that need to be sent to specific ranks. The candidate list to be sent to a specific rank is conservative because any given source cell may not actually intersect any target cell due to our use of bounding boxes of a chosen granularity to represent the target shape. Importantly, we also add cells that are neighbors of the cells in this list based on the requirements of second or higher order remapping algorithms. Such methods need to construct gradients of the numerical field over the source mesh partition and require a complete stencil (set of cells surrounding the cell) for any source cell. Adding the neighbors completes the stencils of the source cells that intersect a target cell on its partition boundary.

Algorithm 2 Overlap Detection

Input: TB: target bounding boxes from all ranks
Output: candidates: list of candidate cells

```
src_tree: Construct the full KD-tree on the source
mesh partition
candidates: list of candidate cells
for r : target ranks do
    for b: TB[r] do
        cells = intersect target bounding box b with
the src_tree
        for c: cells do
            Add c to candidates[r]
            ngbs = find node connected cell neighbors
of the cell c
            Add ngbs to candidates[r]
        end for
    end for
end for
```

3.3 Mesh Migration

After overlap detection, we finally do a mesh migration to send the partial source mesh partition to the required ranks. During overlap detection, the candidate lists can include both owned and ghost cells and we ensure uniqueness of entities on a particular rank after mesh migration. Our mesh migration algorithm is based on a two-pass communication strategy.

1. First pass: We send the number of total counts (owned plus ghost entities) and the number of ghost counts to all ranks using all-to-all communication mechanism. At the end of first pass, all ranks have received the total number of new cells (as well as nodes, topology and numerical fields) they are going to receive on their rank. Based on this information, the receiving data buffers are set to the correct size.

2. Second pass: In this round of communication, we perform a point-to-point blocking send to transmit the actual data, and a non-blocking receive to receive the data from other ranks.

We start by sending the global ids of the candidate source cells on the current rank. After this round of communication, each rank now might have cells with the same global ids, requiring de-duplication. We perform a de-duplication based on the unique global ids so that each entity has only one instance and no duplicate data is stored. We do the same for node global ids as well as other auxiliary mesh entities like edges, faces, etc. We then continue to communicate all the necessary mesh information such as node coordinates, adjacencies such as cell to node connectivity, node to cell connectivity, etc. as well as the numerical fields.

4. NUMERICAL RESULTS

For our numerical studies, we use two sets of geometries. The first shape, shown in Figure 5a, is part of a spherical shell. The second shape, shown in Figure 5b, is of a notional tesseract with six pyramids covering a cube. We chose this shape because while the exterior is a cube, the bounding boxes of the pyramids are highly intersecting. We are trying to represent a worst case example for intersecting partitions. The mesh details for these geometries are provided in Table 1.

We compare the KD-tree method with the bounding box method. Our parameter space for the study consists of:

1. the depth of the KD-tree representation of the target mesh partition, and

2. total MPI ranks (from 2 to 36).

For each point in the parameter space, we collect two pieces of data:

1. the number of new cells received on a rank after redistribution, and

2. the total time to perform the redistribution.

The count of only the new cells provides an approximate estimation of how much extra memory would

Table 1: Mesh Details

Mesh	Source	Target
Sphere	Tetrahedral	Hexahedral
#Cells	1523150	44160
#Points	283924	50952
Tesseract	Tetrahedral	Tetrahedral
#Cells	231828	695805
#Points	45742	130467

need to be stored after redistribution. By varying the number of MPI ranks, we can get very different qualities of partitioning. If the number of ranks respects the symmetries of a mesh we can get a quite good partitioning, but if this is not the case, we can get poor partitioning because cells can be "just stuffed" anywhere. Our study is intended to evaluate all possibilities and not just best case. We use the ParMetis partitioner ([13]) for initial partitioning of both source and target meshes.

4.1 Sphere Shell Mesh

Figures 6a and 6b show the number of migrated cells which is a proxy for the memory estimates of each method after redistribution. The x-axis is the number of ranks on which the test is run, and the y-axis is the depth of the tree. In the waterfall plots, the depth axis has no meaning for the bounding box distributor but we keep it in the figures to make comparisons easier. At each x and y point, we plot the maximum among all the ranks corresponding to the worst case rank. The color map in each plot is a monochromatic palette with the deeper color represent a higher value.

In comparison to the bounding box algorithm, the new method copies significantly fewer source cells to target partitions resulting in substantial reduction in memory usage and network traffic. With increasing depths of the target KD-tree, the target mesh partition representation becomes finer, and as a result the overlap detection improves until it gets to a point where the optimal overlap is detected. We see this behavior in the plot. Each KD-tree representation is a subset of the representation at any coarser depth. Due to this fact, the number of migrated cells is monotonically decreasing with increasing depth. Also note in the figures that there is no data for higher depths and higher ranks. This is because as the number of ranks increase the average number of cells per rank decreases and the maximum depth of the KD-tree on the smaller partitions are less than the maximum depth of partitions on lower number of ranks, so we don't run those cases. Clearly, for a particular number of ranks, increasing the KD-tree depths leads to better overlap detection in comparison to the bounding box algorithm. We also observe that as the number of ranks increase, KD-tree based overlap detection improves the cell counts when compared to bounding box algorithm which does not improve as it is too conservative. With increasing ranks and depths, we see savings around one order of magnitude with the KD-tree algorithm.

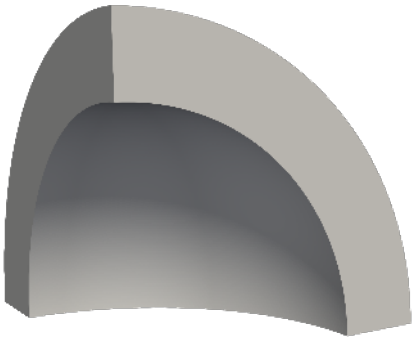
We observe a similar pattern in the time taken by the redistributors as shown in 7a and 7b. The KD-tree algorithm outperforms the bounding box algorithm both in terms of memory savings and time as the number of ranks and depths increase. We also observe a slightly concave pattern with regards to the KD-tree depth, especially on the lower ranks and higher depths, due to increased computation needed for overlap detection.

4.2 Tesseract Mesh

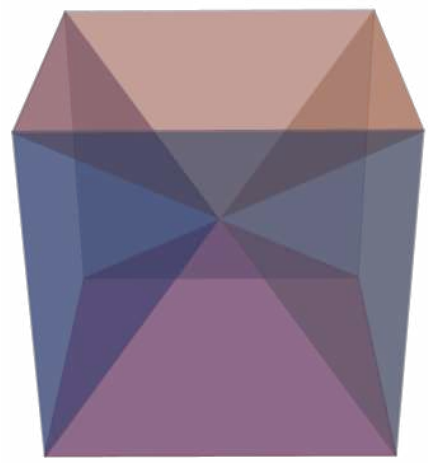
Figures 8a and 8b show the surface plots of the number of migrated cells which is a proxy for the memory estimates and network traffic of each method after redistribution. The surface plot shows a more complex landscape. Repeating what was stated earlier, the tesseract is designed to be representative of a worst case scenario because of the highly intersecting nature of the bounding boxes by construction. Figures 9a and 9b show another view of the same data. Here, we plot the values for all depths at each point on the x-axis.

In comparison to the bounding box algorithm, the new method performs significantly better in terms of memory savings. The target mesh partition representation becomes better resolved with increasing target KD-tree depths, and subsequently the overlap detection becomes optimal after a certain depth. Note both the monotonically decreasing number of migrated cells with increasing depth and the generally improved performance of both algorithms when the number of partitions is a multiple of 6 which is a natural symmetry of the mesh giving "nicer" partitions. We see this behavior in both plots. The bounding box algorithm, on the other hand, does not improve even when the number of ranks is increased, as the bounding box based overlap detection is too conservative. Again, depth has no meaning in the bounding box redistributor. This results in receiving entire source mesh partitions for many ranks where only small pieces are needed. This behavior is due to how the tesseract mesh is partitioned. For example, Figure 10a shows the target part on rank 3 of a four rank run, the source partition is shown in Figure 10b which does not cover the target part at all. This disconnected partitioning is generated by the ParMetis partitioner and is not a pathological construction. As the bounding box (shown in Figure 11a) is the whole cube, clearly all the other mesh parts would be migrated to this rank.

We also observe a similar pattern for the KD-tree al-

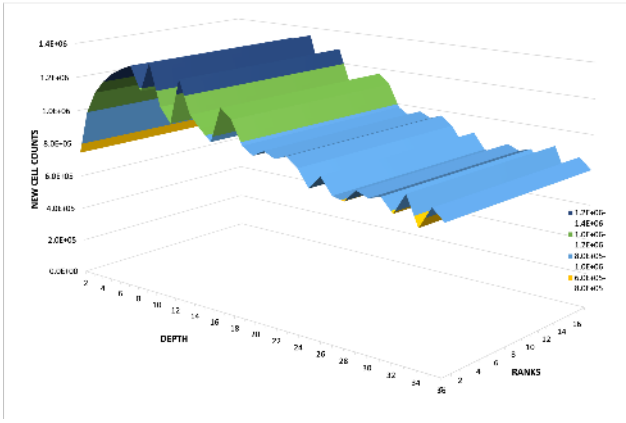


(a) Spherical shell.

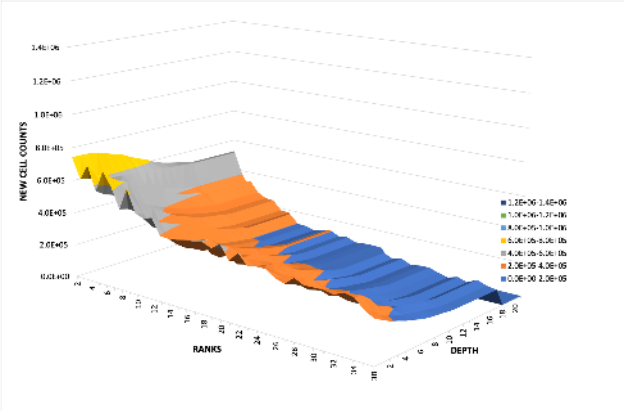


(b) Tesseract with six pyramids covering a cube.

Figure 5: The geometry of two test cases used for the numerical studies.



(a) Counts using the Bounding Box redistributor.



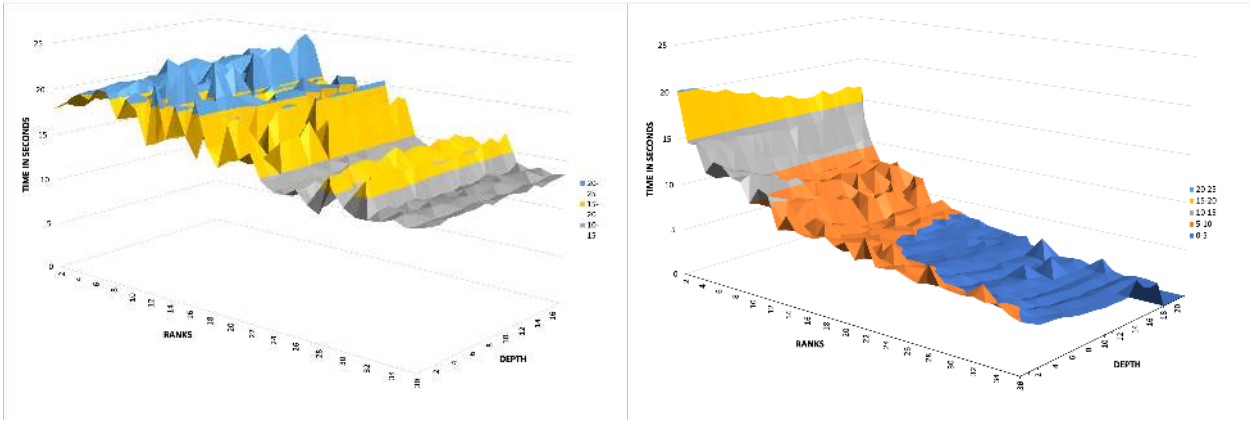
(b) Counts using KD-tree redistributor.

Figure 6: The maximum count of new cells received among all ranks at each number of ranks and for each KD-tree depth of the sphere mesh.

gorithm when the depth is zero, but, as we increase the number of depths, for example as shown in Figure 11b where the bounding boxes of depth 3 tree is overlayed on the target mesh partition, we see significant gains due to better capturing of the target mesh partition shape and finer overlap detection. As we increase the number of ranks and the number of depths, we obtain from 50% reduction to an order of magnitude improved savings.

Figures 12a and 12b show the surface plots of timings of each method after redistribution. Here the timing landscape is complex, and shows concavity. In Figures 13a and 13b, we plot another view of the same data. The bounding box algorithm takes relatively the same amount of time independently of the number of ranks it is run on. This is consistent with the number of new cells received. However, the KD-tree algorithm performance shows greater variability. Overall KD-tree algorithm outperforms the bounding box algorithm both in terms of memory savings and time as the number of ranks and depths increase.

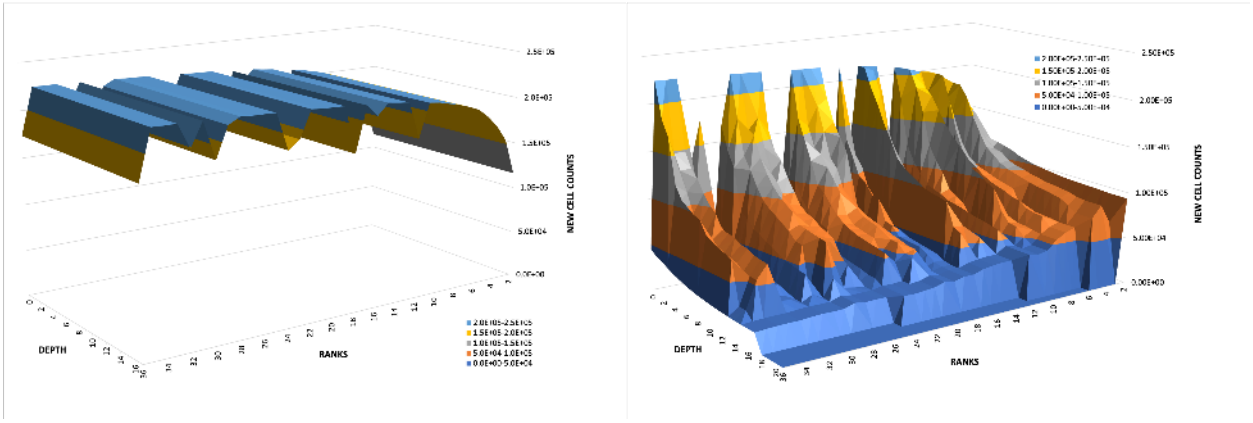
On lower number of ranks, the higher the number of depths, the longer it takes in comparison to the bounding box algorithm. Overall, we observe a concave pattern along the y-axis (the number of depths). Upon investigation, we found the detection was the biggest contributor to the increased timing. As the depths increase, the granularity of the target mesh partition representation also increases. Because the global target mesh is around 700k cells, the overlap detection works with almost that many cells on each rank for higher depths, and thus takes more time. This effect is more prominent on lower ranks as they have significant number of source cells as well to compute



(a) Timing of Bounding Box redistributor.

(b) Timing of KD-tree redistributor.

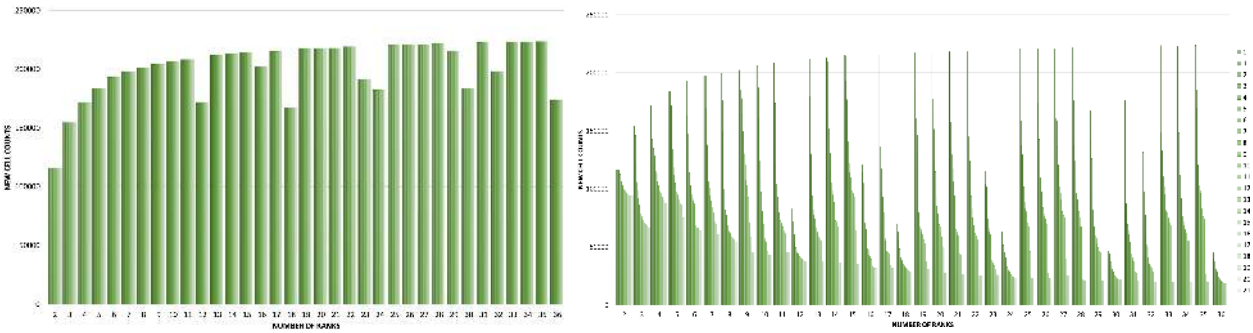
Figure 7: The maximum time across ranks for each KD-tree depth of the sphere mesh.



(a) Bounding Box redistributor.

(b) KD-tree redistributor.

Figure 8: The maximum count of new cells received across ranks for each KD-tree depth of the tesseract mesh shown for both KD-tree redistributor and bounding box redistribution. There the depth do not have any meaning for the bounding box algorithm.

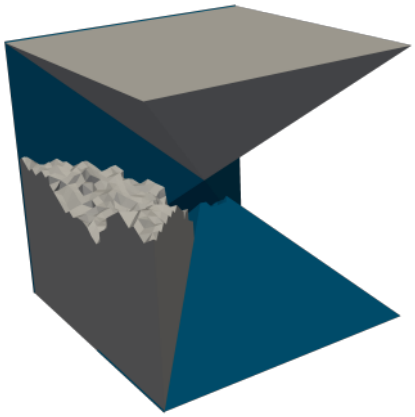


(a) Counts using the Bounding Box redistributor.

(b) Counts using KD-tree redistributor.

Figure 9: The maximum count of new cells received across ranks for each KD-tree depth of the tesseract mesh.

(a) Target mesh partition on rank 3.



(b) Source mesh partition in grey on rank 3.

Figure 10: Target and source mesh partitions on rank three on a four rank partition.

their intersection. We also observed this pattern for the sphere mesh, though not this pronounced as the size of the global target mesh is comparatively small around 44k cells. Finally, as the number of ranks is increased, the KD-tree algorithm becomes comparable or faster than the bounding box algorithm.

5. CONCLUSION

We present a new approach to mesh redistribution for data remapping algorithms. Our method utilizes the KD-tree data structure to improve overlap detection between source and target partitions. We demonstrate the significant savings both in terms of memory and timing by the new algorithm in comparison to the default bounding box algorithm. We observe that, in general, sending the full tree of the target mesh partitions leads to both optimal memory savings and total time to redistribute the mesh. For a coarse target mesh, the shape of the mesh on an individual rank becomes regular and has only a few elements as the number of ranks become large, there probably won't be much gain in describing the geometry using depth zero or the full tree. However, if the target mesh is fine enough so that even on large ranks, the mesh part consists of hundreds to thousands of elements, the over estimation of overlap between target can be significantly reduced by using higher depths. However, if the global target mesh is large, then it might take a lot more time to compute the optimal overlap. In such scenarios, an intermediate depth would perform decently both in terms of memory savings and time.

ACKNOWLEDGMENT

This work is supported by the U.S. Department of Energy for Los Alamos National Laboratory under contract 89233218CNA000001. We thank ASC NGC Ristra and Portage for support.LA-UR-23-21719.

References

[1] Hirt C., Amsden A., Cook J. "An arbitrary Lagrangian-Eulerian computing method for all flow speeds." *Journal of Computational Physics*, vol. 14, no. 3, 227–253, 1974

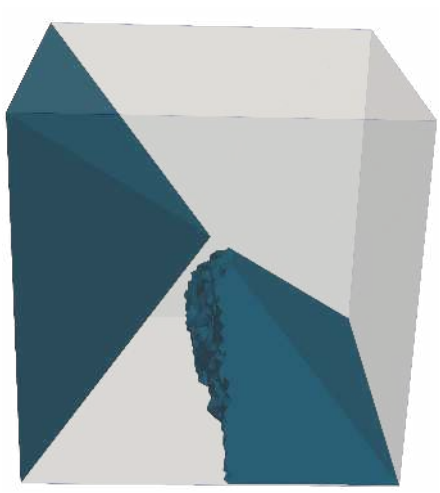
[2] Margolin L., Shashkov M. "Second-order sign-preserving conservative interpolation (remapping) on general grids." *Journal of Computational Physics*, vol. 184, no. 1, 266 – 298, 2003

[3] Barlow A.J., Maire P.H., Rider W.J., Rieben R.N., Shashkov M.J. "Arbitrary Lagrangian–Eulerian methods for modeling high-speed compressible multimaterial flows." *Journal of Computational Physics*, vol. 322, 603–665, 2016

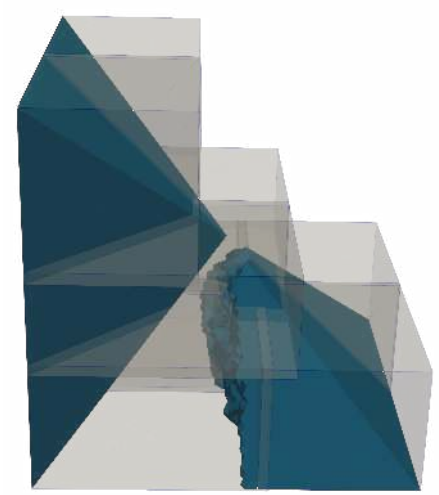
[4] Kucharik M., Breil J., Galera S., Maire P.H., Berndt M., Shashkov M. "Hybrid remap for multi-material ALE." *Computers & Fluids*, vol. 46, no. 1, 293–297, 2011

[5] Robinson A., Brunner T., Carroll S., Drake R., Garasi C., Gardiner T., Haill T., Hanshaw H., Hensinger D., Labreche D., et al. "ALEGRA: An arbitrary Lagrangian-Eulerian multimaterial, multiphysics code." *46th AIAA Aerospace Sciences Meeting and Exhibit*, p. 1235. 2008

[6] Painter S.L., Coon E.T., Atchley A.L., Berndt M., Garimella R., Moulton J.D., Svyatskiy D., Wilson C.J. "Integrated surface/subsurface permafrost thermal hydrology: Model formulation

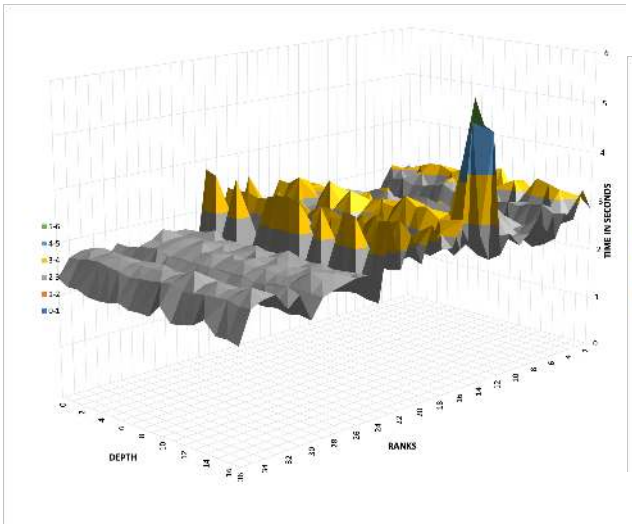


(a) Bounding box corresponding to depth 0.

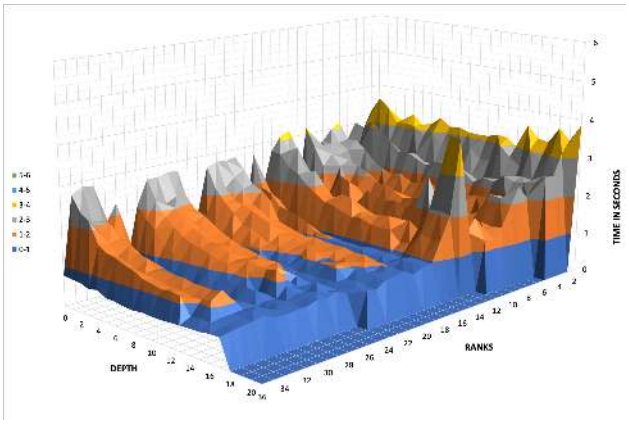


(b) Bounding box corresponding to depth 3.

Figure 11: Difference in target description based on KD-tree depths.

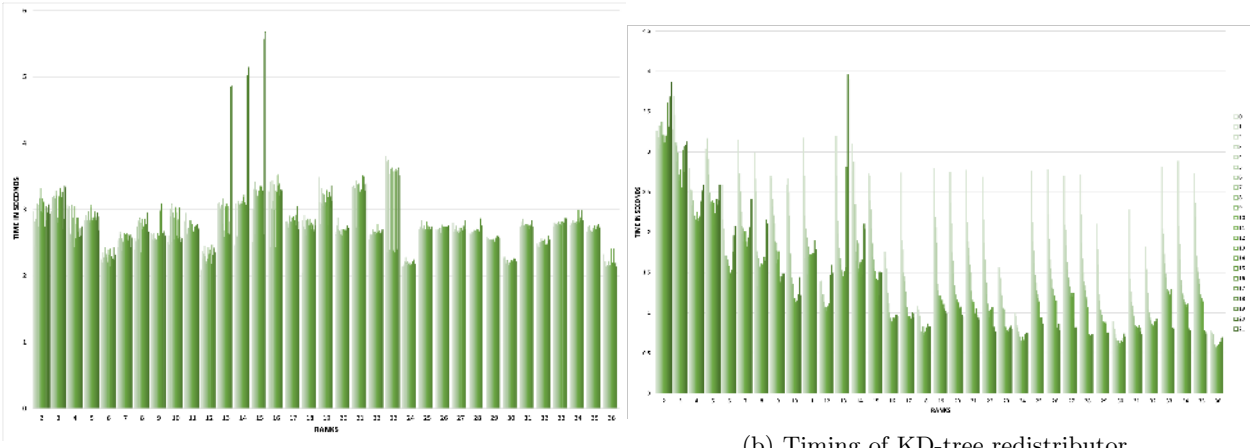


(a) Timing of Bounding Box redistributor.



(b) Timing of KD-tree redistributor.

Figure 12: The maximum time across ranks for each KD-tree depth.



(a) Timing of Bounding Box redistributor.

(b) Timing of KD-tree redistributor.

Figure 13: The maximum time across ranks for each KD-tree depth.

and proof-of-concept simulations." *Water Resources Research*, vol. 52, no. 8, 6062–6077, 2016

[7] Burton D.E. "Lagrangian hydrodynamics in the FLAG code." *Los Alamos National Laboratory, Los Alamos, NM, Technical Report No. LA-UR-07-7547*, 2007

[8] Herring A., Ferenbaugh C., Malone C., Shevitz D., Kikinzon E., Dilts G., Rakotoarivelo H., Velechovsky J., Lipnikov K., Ray N., et al. "Portage: A Modular Data Remap Library for Multiphysics Applications on Advanced Architectures." *Journal of Open Research Software*, vol. 9, no. 1, 2021

[9] Plimpton S.J., Hendrickson B., Stewart J.R. "A parallel rendezvous algorithm for interpolation between multiple grids." *Journal of Parallel and Distributed Computing*, vol. 64, no. 2, 266–276, 2004

[10] Slattery S.R., Wilson P.P.H., Pawlowski R.P. "The Data Transfer Kit: A geometric rendezvous-based tool for multiphysics data transfer." *American Nuclear Society*, 7 2013. URL `https://www.osti.gov/biblio/22212795`

[11] Zhang J., Guo H., Hong F., Yuan X., Peterka T. "Dynamic Load Balancing Based on Constrained K-D Tree Decomposition for Parallel Particle Tracing." *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, 954–963, 2018

[12] Bentley J.L. "Multidimensional Binary Search Trees Used for Associative Searching." *Commun. ACM*, vol. 18, no. 9, 509–517, sep 1975

[13] Karypis G. *Encyclopedia of Parallel Computing*, chap. METIS and ParMETIS, pp. 1117–1124. Springer US, Boston, MA, 2011