

Space-filling curves for Partitioning Adaptively Refined Meshes

Aparna Sasidharan* and Marc Snir[†]*

*Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL

Email: sasidha2@illinois.edu

[†]Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL

Email: snir@anl.gov

Abstract—We present a general space-filling curve algorithm for partitioning an arbitrary 3D mesh. We discuss communication patterns in Adaptive Mesh Refinement (AMR) applications and how we can reduce communication and improve the quality of partitions using a better space-filling curve. We compare our partitions with those generated using Morton order, which is currently used by majority of AMR software frameworks. We used the MiniAMR miniapp from Mantevo to generate our test cases and also measure the various costs involved in adaptive mesh refinement.

Index Terms—Space-filling Curve; Mesh partitioning; Geometric partitioning; AMR; Load balancing

I. INTRODUCTION

Adaptive Mesh Refinement is a general technique used for solving problems in science and engineering where the amount of computation can be significantly reduced by adjusting the precision of the solution according to the evolution of the computation. The governing equations are solved on a discrete mesh with varying resolution. A fine mesh is used in the regions that are more sensitive to resolution (e.g., because of turbulence), and a coarse mesh is used to cover the rest of the domain. This reduces both computation and storage, as compared to a mesh with fixed resolution.

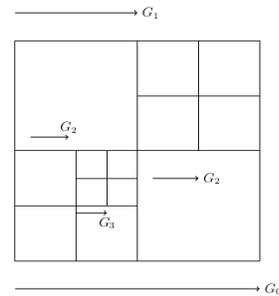
The locations of the regions of interest in an AMR application change during the course of the simulation and the mesh is adaptively refined or coarsened according to location changes. This creates an interesting and challenging problem for parallel computing that involves repartitioning a mesh on the fly with as little overhead as possible. The new partitions should satisfy the criteria of a *good* partition i.e load balance and minimum communication, as much as possible. Repartitioning a mesh during the course of a simulation involves several additional costs besides the actual partitioning phase. It includes migrating data to its new location and recomputing the communication pattern for the next phase of the simulation. A good repartitioning scheme should minimize data migration and generate the new communication pattern quickly. Space-filling curves (SFC) have been used as a preferred partitioning scheme for AMR applications due to their low overhead, good

load balance and low data migration costs. However, not much work has been done on selecting SFC partitions that are most appropriate for the dynamic, irregular meshes used in AMR.

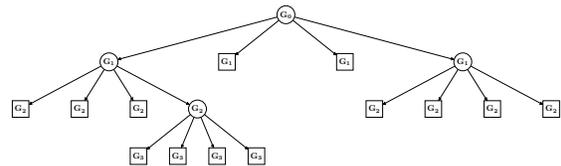
II. THE ADAPTIVE MESH REFINEMENT PROBLEM

AMR applications can be broadly divided into Structured AMR (SAMR) and Unstructured AMR (UAMR). In this work, we have only evaluated SAMR, although SFCs can be used for partitioning UAMR problems as well.

A. Structured AMR - Representation



(a) Adaptively Refined Mesh



(b) Grid Hierarchy

The grid cells in SAMR are polygons bounded by hyperplanes parallel to the dimensions of the domain i.e rectangles in 2D and rectangular boxes (or cuboids) in 3D. The numerical methods for solving these problems were originally defined by [1]. There are two popular implementations of this method - *patch-based* and *tree-based*. Both implementations

maintain a hierarchy of grid levels, with grid cells at the same refinement level appearing at the same height of the hierarchy tree. The levels are nested, i.e. grid cells at level $l + 1$ are formed from subsets of those at level l which satisfy certain refinement criteria. The patch-based scheme groups the refined cells at a level into mostly non-overlapping patches of different sizes. The tree-based scheme creates equal-sized blocks out of the grid cells at a level. This leads to slightly different requirements for each implementation. Patches are typically larger than blocks. The patch-based method is more constrained by the load imbalance of various sized jobs. The tree-based scheme on the other hand needs better partitioning schemes that can reduce communication during refinement and halo-exchange. In this paper, we have only dealt with the tree-based AMR implementation.

1) *Tree-based SAMR*: Figure 1a shows an adaptively refined mesh in 2D with two levels of refinement. The refinement ratio is 2 in either dimension. The grids are named according to their refinement levels. G_0 corresponds to a coarse block covering the entire problem domain. It is refined into four blocks labelled G_1 . Two of the G_1 blocks are further refined to create blocks belonging to G_2 and so on. Figure 1b has the corresponding tree. The non-terminal nodes in the tree are indicated using circles and terminals (leaf nodes) are represented by squares. In a tree-based implementation, the AMR solution is computed only at the terminal nodes of the tree. The AMR scheme allows for refinement in space as well as time. If ref_ratio is the refinement ratio, a refined block has $\frac{1}{ref_ratio}$ times the resolution of its parent block in each dimension. Similarly, the refined blocks have the option to advance the solution by ref_ratio timesteps compared to its immediate coarse neighbor. The communication costs in SAMR can be broadly classified into the following:

- Intra-level:** This is the cost of halo-exchange between neighbor blocks that are at the same level of refinement.
- Inter-level:** A block at level l may share a boundary with a coarse block at level $l - 1$ or with finer blocks at level $l + 1$. Data is exchanged between these boundary blocks using a scatter-gather pattern. If refinement is done in time, the coarse-fine boundaries need to be synchronized.
- Refinement/De-refinement:** The decision to refine or de-refine a block requires communication between itself and its parent and other neighbor blocks.

The first two costs require the partitions to have good *locality*, i.e. blocks that are physically close in space should be located on the same processor as far as possible. The third cost includes communication across the tree hierarchy. This depends to a large extent on how the distributed tree data structure is implemented. Ideally, a non-terminal block should be co-located with its child blocks to minimize this communication.

We have measured only the cost for intra and inter-level communication for comparison: The code we used, namely the MiniAMR code from the Mantevo suite [2], is not designed so as to localize structures at different levels of the tree, so that communication is not improved with a better partition. Also, we have only considered refinement in space; all active blocks advance the solution at the same rate.

III. SPACE-FILLING CURVES IN 3D

A *Space-Filling Curve* (SFC) f in d -dimensional space is defined as a continuous surjective function from the interval $[0, 1]$ to $[0, 1]^d$. Space Filling Curves are usually built through successive approximations by piecewise linear curves that connect the k^d points of a rectangular grid of equally spaced points in $[0, 1]^d$, for increasing values of k . Each curve is non-intersecting and traverses each point once. We call these approximations *Finite Space Filling Curves*, or just Space Filling Curves, when the meaning is clear from the context. Such a finite SFC can be identified with a linear order on the grid points: the curves order the points and is obtained by connecting successive points in the linear order. These curves are *locality preserving*: successive points on a curve are grid neighbors; more generally, m successive points are contained in a cube of volume $O(m/k^d)$.

These curves can be used to partition rectilinear grids: One uses a curve that snakes through the center of the bricks. The elements are partitioned into p subsets by cutting the curve into p segments containing each roughly the same number of points. The partitions have good locality, so that, if the computation requires communication between adjacent bricks, the partition will be load-balanced and have low communication.

Some of the popular SFCs include Hilbert, Peano and Sierpinski [3]. Although Morton order [4] does not fit into the definition of being derived from a family of finite space filling curves, it is widely used due to its ease of computation. A Morton order of points in R^d can be generated by interleaving the binary representation of the co-ordinates of the points, followed by sorting.

Hilbert curves are well-defined in 2D; there is a unique finite Hilbert curve for any $2^n \times 2^n$ points on a plane. However, the definition is not clear for dimensions > 2 . Sagan [5] provided a mathematical and geometric interpretation of a Hilbert curve in 3D, but this definition is not unique. Also, SFCs are generally defined only for grids which have sizes that are powers of 2 (Hilbert) or 3 (Peano). These constructions can be generalized to meshes with other dimensions e.g., by embedding them into meshes of the required dimensions. Tree-based SAMR meshes can be handled by using a curve (Hilbert for a refinement of 2, Peano for a refinement of 3) that snakes through all cells at the highest level of refinement. Subcells of a coarser cell are traversed consecutively, so that the curve also traverses each of the cells of the adaptive mesh once. However, these methods result in partitions with degraded locality.

By extension, a Space-Filling Curve for a set S of points in \mathbb{R}^d is a continuous, non-intersecting curve that passes through each point once and is locality-preserving. (We do not provide

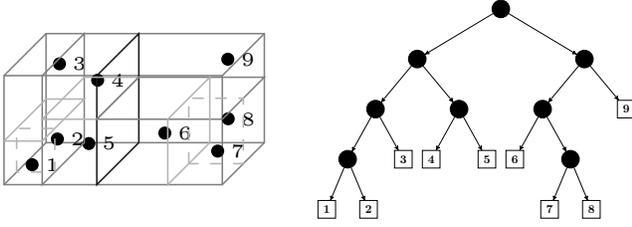


Fig. 1. kd-tree for a set of 9 points

a formal definition of locality-preserving for the general case, but use it loosely to mean that successive points on the curve tend to cluster in space.)

Such a space filling curves can be used to partition irregular meshes in \mathbb{R}^d : Each cell is represented by a point in the cell, such as its center of gravity; a SFC is used to partition the points and thus partitioning the cells. If the cells have good quality shapes then the resulting clusters have good surface to volume ratios, resulting in low communication overheads.

In the next section we discuss a recursive algorithm to generate good space-filling curves for arbitrary sets of points in 3D. Our definition is independent of the geometry of the domain and hence suitable for adaptively refined meshes.

IV. THE 3D GENERAL SFC ALGORITHM

There are two general techniques to create a finite space filling curve for a set of points in \mathbb{R}^d

- 1) Bit-manipulation of the co-ordinates of the points to generate keys, followed by sorting of the keys
- 2) Recursion.

We used recursion to define our SFC. We first arrange the points in a kd-tree and then traverse it based on a set of rules to generate the curve. A kd-tree is a data structure commonly used for solving multi-dimensional search problems efficiently [6]. Each node of the tree corresponds to a cuboid in 3D. The root node of the tree represents a cuboid containing all the points. Subsequent sub-domains (nodes in the tree) are created by recursively splitting along a hyperplane perpendicular to one of the dimensions until further splitting is not possible, i.e there is only one point at a node. These form the leaf nodes of the kd-tree. In a typical construction, the splitting is done by alternating between the three co-ordinate dimensions and the hyperplane passes through the midpoint of the splitting dimension. For all of our test cases, we built the kd-tree by splitting along the dimension of maximum spread of the points. The hyperplane was positioned either at the midpoint of the splitting dimension, splitting the box into sub-two boxes of equal length; or at the median, splitting into two sub-boxes containing an equal number of points. Figure 1 shows an example for kd-tree construction for a set of 9 points in 3D. In this example, we always split along the median of the dimension of maximum spread.

A. Traversal Rules

Once the kd-tree is constructed, the SFC is generated by traversing the tree according to a set of traversal/refinement

rules: For each cuboid, we specify the face through which the curve enters the cuboid and the face through which it exits it. The order of traversal of the leaves of the kd-tree specifies the SFC. The traversal rules are also defined recursively. We start by specifying an order for the root node. The orders of traversal for the children of a node are generated by refining the order for the parent node when it is split.

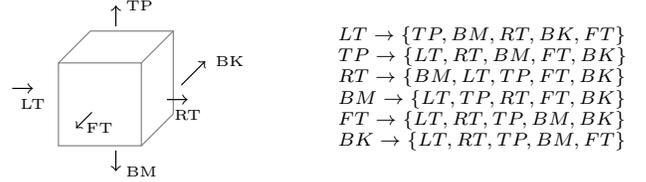


Fig. 2. Directions for entry/exit for each face of a cuboid

Our rules generate a hilbert-like curve in 3D that is *face-continuous* - adjacent cuboids share a common face and the curve use this common face to traverse from one cuboid to the next. As shown in figure 2 there are 30 different choices of entry and exit faces for a 6-faced cuboid. Most of these are symmetric transformations of each other. We need to consider only two base cases - *cis* and *trans*. *Trans* considers the scenario where the curve enters and leaves the cuboid through opposite parallel faces. *Cis* covers all cases where the curve enters and exits the cuboid through its adjacent faces (sharing a common edge). All traversal rules can be generated by applying symmetric transformations to the rules for these two base cases.

The *cis* case has three sub-cases based on whether the cuboid is split in a dimension parallel to the entry face, parallel to the exit face or perpendicular to both. There are unique ways to traverse the sub-domains for the first two cases. For the third case, we chose one of the two options that minimizes discontinuity in the curve by matching the entry location with the previous exit face. Figure 3 describes the three cases in detail.

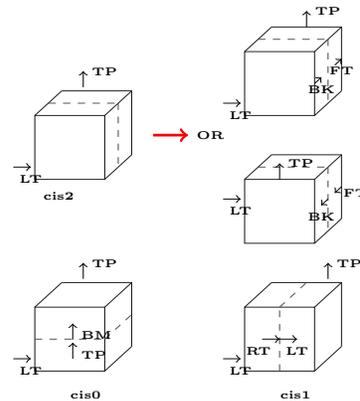


Fig. 3. cis base rule

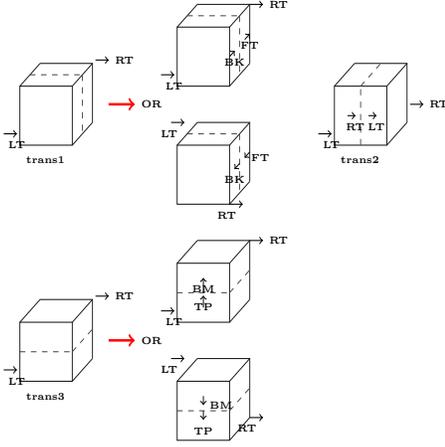


Fig. 4. trans base rule

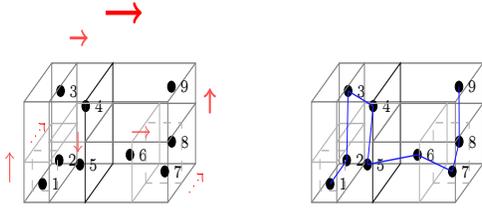


Fig. 5. SFC on a set of 9 points in 3D

Similarly, for *trans*, one can identify three cases based on whether the splitting dimension is parallel to the entry and exit faces or perpendicular to both. We deal with ambiguities in the same manner as before, by choosing the shortest face-continuous path through the sub-domains. The *trans* rules are explained in figure 4.

Figure 5 provides an example for traversing a kd-tree of 9 points in the *left to right* direction. The cuboid on the left has the directions marked on it in red, based on the traversal order of the nodes in the tree. The cuboid on the right has the generated curve drawn in blue. We have an example for resolving an ambiguity here. After visiting point 6, the curve could either visit point 7 or point 8. But we choose point 7 because it is closer to 6 and therefore the shortest path for the curve. The pseudocode for kd-tree traversal is provided in algorithm 1.

B. Special Cases

We described the construction assuming that each box is split into two sub-boxes when refined. Many SAMR algorithms refine boxes by directly splitting them into four quadrants in 2D, or eight octants in 3D. This types of refinement can be handled as special cases of the general algorithm.

The general algorithm has the following degrees of freedom:

- The order in which boxes are split
- The choice of the dimension being split

Algorithm 1 Kd-tree Traversal

```

procedure TRAVERSE(n)
  entry ← n.ENTRY()
  exit ← n.EXIT()
  splitdim ← n.SPLITDIM()
  entry_pt ← n.ENTRY_LOC() ▷ Used to resolve ambiguities
  first ← ORDER(entry, exit, splitdim, entry_pt) ▷ Selects
  which child is visited first
  u ← n.CHILD(first)
  v ← n.CHILD(1-first)
  u.entry ← entry
  v.exit ← exit
  if splitdim = dimX then
    if first = 0 then
      u.exit ← RT
      v.entry ← LT
    else
      u.exit ← LT
      v.entry ← RT
    end if
  else if splitdim = dimY then
    if first = 0 then
      u.exit ← BK
      v.entry ← FT
    else
      u.exit ← FT
      v.entry ← BK
    end if
  else
    if first = 0 then
      u.exit ← TP
      v.entry ← BM
    else
      u.exit ← BM
      v.entry ← TP
    end if
  end if
  if u.size() > 1 then
    TRAVERSE(u)
  end if
  if v.size() > 1 then
    TRAVERSE(v)
  end if
end procedure

```

- The choice of the displacement for the splitting hyper-plane.

It is easy to see that refinements into quadrants or octants can be obtained by a suitable choice of these parameters.

V. PARALLEL ALGORITHM FOR SFC CONSTRUCTION

The tree-based AMR is naturally suited to a parallel implementation of SFC ordering on the boxes. The domain is discretized into boxes which have a strict hierarchical relationship between each other. The active boxes in the domain are the leaf nodes in the tree. New boxes are created during the refinement stage by splitting an existing active box. The box that is split is no longer an active box and is promoted to being a parent node in the tree. Boxes are deleted from the tree during the de-refinement or coarsening stage. A parent node in the tree becomes the new active box and its child boxes are removed during coarsening.

We use the co-ordinates of the center of a box as the data points in our SFC algorithm. Each box is associated with an entry face and an exit face. When a box is split into two sub-boxes then the algorithm described in the previous section is used to compute the entry and exit faces for each of the two sub-boxes. In particular, this determines the order in which the two sub-boxes are visited (they are visited consecutively). We also generate recursively keys that encode this traversal order.

The parallel SFC algorithm has two steps :

- 1) Key Generation
- 2) Sorting

A. Key Generation

This phase of the algorithm assigns keys to newly added blocks to the tree. Each node is associated with a binary string. The root is associated with the empty string \perp . If a box associated with the key k is split into two sub-boxes, then the first traversed box is associated with the string $k0$ and the second traversed box is associated with the string $k1$. When the sub-boxes are merged back, then the parent box regains its original key k . It is easy to see that the lexicographic order of the keys encodes the SFC traversal order.

In practice, a key $k = k_1 \dots k_m$ is represented by the integer number $k_1 \dots k_m 0 \dots 0$. In order to properly compute key values one also needs to store the key length, i.e., the depth of the node in the tree.

We can use several shortcuts: If the computation starts from a regular $2^k \times 2^k \times 2^k$ Cartesian mesh then the keys for the mesh boxes can be computed directly from the box indices. If boxes are split into octants, then we can compute directly the keys of the eight children from the key of the parent.

B. Sorting

Once the keys are generated, we sort them locally. If the initial domain was ordered and partitioned according to some SFC, then concatenating the partially sorted key lists is sufficient to generate the full curve. All the keys on processor P_i will be greater than or equal to those on P_{i-1} and less than or equal to the keys on P_{i+1} . Now, the curve needs to be

sliced so that all processors have roughly the same workload. Since sorting is local, we need to determine the position of a block/key in the global order. This is done using a parallel prefix operation to determine the total number of keys to the left of a particular key. After this step, the curve can be sliced into equal sized segments. For the test cases we used in this paper, all blocks performed the same amount of work. In the case where the mesh is refined in time, we can add a weight value to a block that is inversely proportional to its refinement level (since finer blocks are updated more often). The curve should then be sliced into equal weight segments. Paramesh [7] uses a similar technique to order the blocks using Morton. But they use bit-interleaving to generate the Morton keys. We wanted a mechanism that is more general and can be applied to irregular distributions.

VI. EXPERIMENTS

We used MiniAMR from the Mantevo miniapp suite to test the quality of our SFC partitions. This miniapp has a tree-based AMR implementation modeled on Paramesh. It was designed with the intention to better understand the communication behavior of AMR applications. Therefore, the computation kernel is very simple. It is a single loop that averages the values of each variable based on a stencil. The baseline implementation of MiniAMR does the following :

- 1) - Create an initial 3D Cartesian grid of blocks and a set of objects in the grid. The position, size and shape of the objects can be decided by the user. The objects can be made to move at a certain speed in the domain. They can also be made to grow/shrink during the course of the simulation. Blocks are refined/de-refined based on whether they intersect the object or not.
- 2) - Refinement/de-refinement decisions are subject to the constraint that no two adjacent blocks should differ by more than one refinement level. The maximum number of refinement levels is decided by the user.
- 3) - After every refinement/de-refinement, the average and maximum load on any processor are computed. Load balancing is triggered if they differ by more than a ratio decided by the user. The current load balancing scheme in MiniAMR uses an RCB (recursive co-ordinate bisection) algorithm that recursively distributes the extra blocks on a processor along each dimension. The blocks are distributed to their new locations, the pointers in the tree are adjusted and the new communication pattern is determined.
- 4) The stencil computation and halo-exchange routines use the current assignment of blocks to processors until the next load balancing step. The simulation runs until the maximum number of time steps is reached. Refinement is done only in space; all active blocks advance at a uniform rate.

The baseline implementation uses separate data structures to store the non-terminal and terminal nodes in the tree. The terminal nodes (active blocks) are not stored in contiguous memory locations, which affects the efficiency of the stencil

computation code as well as the on-processor copying of ghost (halo) data. We made changes to some of the communication routines of the baseline code to improve its performance. The current version of MiniAMR orders inter-processor communication according to the dimensions of the grid; all neighbors in the x dimension are exchanged before y and then z . This was inefficient for SFC partitions. So we aggregated messages so that any pair of processors communicate at most once during a halo exchange step. The parameters which affect the communication time are therefore the maximum degree (the maximum number of distinct messages sent/received by a process) and the maximum communication volume (the maximum number of bytes sent/received by any process).

We also modified the block re-distribution routines of MiniAMR. The baseline implementation did not aggregate blocks during re-distribution - blocks were sent to their new locations one at a time. We aggregated messages so that all the blocks that need to be moved from P_i to P_j are packed and sent in a single message. Besides, we used separate ghost buffers for each face of a block and parallelized on-processor communication using openmp [8].

A. Comparison with Baseline Implementation

#procs	T_{comp}	T_{comm}	T_{lb}	T_{exec}
128	15.24	30.15	3.979	54.70
256	13.10	26.96	4.852	51.17
432	10.74	23.32	6.79	49.35
512	10.40	22.86	5.77	46.72
1024	8.55	19.21	7.15	44.16
2048	7.58	17.98	9.23	47.04

TABLE I
TOTAL RUNNING TIMES FOR THE BASELINE IMPLEMENTATION FOR DIFFERENT PROCESSOR COUNTS

We refer to our SFC as GenSFC for the rest of the discussion. The testcase we constructed for this comparison is that of a deforming sphere hitting a cuboid. Blocks are refined at the interface of the sphere and the cuboid. We ran the simulation for a total of 2000 timesteps with refinement/de-refinement frequency equal to 3 timesteps. Each block consists of $4X4X4$ grid cells and each cell has 32 data variables associated with it. We used a $7-pt$ stencil with a ghost region of $width = 1$ grid cell in all dimensions. Load balancing was done aggressively, i.e whenever the maximum number of blocks on any processor is at least two more than the average. We split total running time into the following costs :

$$T_{exec} = T_{comp} + T_{comm} + T_{lb} + T_{ref}$$

where T_{exec} is the total running time, T_{comp} is the total time for stencil computation and T_{comm} is the time taken for exchange of ghost values over all timesteps. T_{lb} is the total time spent in load balancing. This includes the cost of computing new partitions and re-distributing blocks. T_{ref} is the time taken for refinement and de-refinement. Table I shows the running times for the baseline implementation. Tables II

and III are the running times for the modified miniapp using Morton order and our SFC partitions. The values reported in tables I, II and III are the maximum of the sum over 2000 timesteps.

#procs	T_{comp}	T_{comm}	T_{lb}	T_{exec}
128	12.90	12.74	3.48	35.02
256	11.08	11.28	4.99	35.17
432	9.06	9.44	6.02	33.29
512	8.78	9.11	5.61	32.41
1024	7.21	7.84	6.61	31.62
2048	6.47	7.56	7.20	33.98

TABLE II
TOTAL RUNNING TIMES FOR THE MODIFIED MINIAPP WITH MORTON SFC PARTITIONS FOR DIFFERENT PROCESSOR COUNTS

#procs	T_{comp}	T_{comm}	T_{lb}	T_{exec}
128	12.91	12.81	3.91	35.97
256	11.10	11.03	5.02	35.19
432	9.06	9.13	6.64	33.86
512	8.78	8.93	5.68	32.57
1024	7.22	7.40	6.66	31.37
2048	6.47	6.95	7.99	34.24

TABLE III
TOTAL RUNNING TIMES FOR THE MODIFIED MINIAPP WITH GEN SFC PARTITIONS FOR DIFFERENT PROCESSOR COUNTS

From the observations, it can be seen that message aggregation makes a big difference to the total communication time. Also, the use of openmp for computation and on-processor communication is an added advantage. The load balancing time of the RCB algorithm seems to worsen with increasing number of processors. This is due to the lack of message aggregation during block redistribution. The modifications made to the baseline implementation have optimized the inter-processor communication of the miniapp and this is reflected in the total running times. The GenSFC algorithm we used for this test case uses the midpoint of the dimension of maximum spread as the splitting strategy for the initial domain. Between Morton order and GenSFC, the Morton order partitions have higher T_{comm} due to discontinuities in the curve. GenSFC has slightly higher values for T_{lb} in some cases. This is due to the increased locality of the curve. It places more blocks that are near each-other in physical space on the same processor. This increases the likelihood of co-located blocks being refined and hence more blocks have to be moved during load balancing than Morton. However, the values reported here are an upper bound for T_{lb} since load balancing is done at the slightest change in the number of blocks. In real applications, load balancing is done less often. Also, the observations for 128 processes is interesting. The communication cost for GenSFC is higher than Morton for this case. This is again due to its higher locality. For 128 processes, the communication cost increased as a result of the on-processor memory accesses for copying ghost values. The non-contiguous block data structure used by MiniAMR is the reason behind this increase. The active blocks in the tree should be placed in contiguous

memory locations by compressing the sparse array. This effect does not appear in the remaining observations due to fewer blocks per processor.

Since we have not optimized the refinement/de-refinement communication in MiniAMR, the values for T_{exec} do not reflect the time saved due to better partitions. In the current version of MiniAMR, the communication during refinement is a huge overhead because of separate decompositions used for the non-terminal and terminal nodes and indirect memory accesses. Therefore, we cannot conclude anything from the measured values of T_{exec} . As explained earlier, T_{exec} can only be improved by using better data structures and optimizing communication across the tree hierarchy.

We analyse these results in detail in the following sections using different test cases and splitting strategies for GenSFC. We used the following communication model for our experiments :

$$T_c = degree * a + b * comm_vol$$

where *degree* is the number of distinct messages and *comm_vol* is the total bytes sent/received by a process. *a* is the latency or set up cost of a message and *b* is $\frac{1}{bandwidth}$. The values of T_{comm} reported in our results includes the following costs

- 1) Inter-processor halo exchange
- 2) Packing ghost data into send buffers
- 3) On-processor copying of ghost data (overlapped with inter-processor communication)
- 4) Unpacking of ghost data from receive buffers

B. Testcase1

The first test case resembles an explosion; we created an expanding object inside a cuboid. Refinement was done only at the advancing front of the object. The center of the object was chosen to be the geometric center of the domain (cuboid). The rate of expansion of the object was 2×10^{-3} in all dimensions. We introduced asymmetry in this test case by varying the aspect ratio of the cuboid. The simulation was run for a total of 1000 time steps, the maximum number of refinement levels was chosen to be 6. Refinement/de-refinement was done every 3 time steps. Tables IV and V show the measured values for the maximum of the sum of T_{comp} , T_{comm} , T_{lb} and T_{exec} over all time steps, across all processes. We used blocks of size $4X4X4$ with halo width equal to 1. Also, we have assumed a $7-pt$ stencil for all three test cases.

Table IV gives the values for Morton partitions and table V has the same observations using GenSFC partitions. For the GenSFC algorithm, we used the geometric midpoint of the dimension of maximum spread as the splitter. The initial distribution of blocks was partitioned using serial versions of Morton and GenSFC; the top level keys were assigned during this step. For the subsequent load balancing steps, we used the parallel versions of both curves. Load balancing was done whenever there was a change in the load i.e the maximum number of blocks on any process exceeded the average value

by at least 2. This is a measure of the worst total T_{lb} for both SFCs. The experiments are tabulated as two categories based on message lengths - short and long. When the messages are short the communication time is dominated by the maximum degree of any partition and when they are long, the bottleneck is the bandwidth. The short messages have 2 variables per grid cell and the long messages have 20. We have designed the experiments in such a way that they are weak scaling in some sense - the maximum number of blocks per process per time step was kept approximately the same.

#procs	init_domain	Short Msgs				Long Msgs			
		T_{comp}	T_{comm}	T_{lb}	T_{exec}	T_{comp}	T_{comm}	T_{lb}	T_{exec}
512	32X4X4	0.306	1.81	1.85	21.17	2.66	3.45	3.39	28.03
1024	32X8X4	0.243	1.817	1.20	19.94	2.06	2.56	2.65	26.20
2048	32X8X8	0.514	2.34	3.32	15.52	4.72	5.98	6.81	27.11
4096	32X16X8	0.413	2.09	3.49	15.58	3.81	5.48	7.31	26.25
8192	64X16X8	0.389	2.15	4.85	17.17	3.57	5.34	9.39	28.30

TABLE IV
TOTAL RUNNING TIMES FOR TESTCASE1 WITH MORTON PARTITIONS FOR DIFFERENT PROCESSOR COUNTS

#procs	init_domain	Short Msgs				Long Msgs			
		T_{comp}	T_{comm}	T_{lb}	T_{exec}	T_{comp}	T_{comm}	T_{lb}	T_{exec}
512	32X4X4	0.306	1.04	1.86	20.13	2.66	3.04	3.39	27.13
1024	32X8X4	0.243	0.89	1.61	20.89	2.06	2.32	3.74	27.11
2048	32X8X8	0.514	1.91	3.4	15.02	4.72	5.46	6.78	26.87
4096	32X16X8	0.413	1.69	3.61	15.12	3.81	4.83	7.38	25.92
8192	64X16X8	0.389	1.81	4.83	16.83	3.57	4.78	9.41	27.81

TABLE V
TOTAL RUNNING TIMES FOR TESTCASE1 WITH GENSF (MIDPOINT) PARTITIONS FOR DIFFERENT PROCESSOR COUNTS

The observations in tables IV and V show the lower T_{comm} for the GenSFC partitions, for both short and long messages. The discontinuities in Morton order increase its maximum degree as well as maximum communication volume. The difference is more significant at large process counts when there are few blocks per process and when the domain is asymmetric. We designed our experiments carefully to capture this difference. The worst case T_{lb} of GenSFC partitions is slightly worse due to its better locality. But this is an upper bound, which can of course be amortized by load balancing less frequently. The improved communication times does not reflect adequately in the total execution time due to the same reasons explained in the previous section.

C. Test case2

This test case was designed to closely represent slowly moving particle distributions in astrophysics simulations. The distributions tend to display clustering of points in different regions of the domain as the simulation evolves. We created an ellipsoid object with a maximum of 4 levels of refinement throughout its volume positioned inside an asymmetric cuboid using MiniAMR. The object was initially positioned near the left face of the cuboid and made to move slowly towards the right with a constant velocity of the order of microseconds. The simulation was run for a total of 2000 time steps. The refinement/de-refinement frequency was set to 3 time steps. Block size, ghost region width and stencil are same as the

previous test case. We measured the total computation time, communication time, load balancing time and execution time for Morton and GenSFC partitions. For the GenSFC algorithm we experimented with two splitting functions

- 1) Geometric midpoint of the dimension of maximum spread
- 2) Median of the point distribution along the dimension of maximum spread

We used a serial version of Morton and GenSFC for the initial set of blocks, followed by parallel versions of the curve for the subsequent load balancing steps.

Figure 6 and figure 7 show the curves generated for this testcase for 128 processors and 3 levels of refinement.

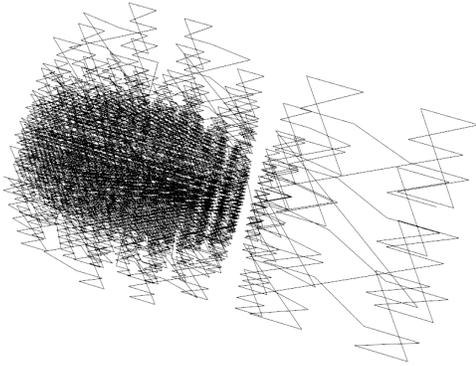


Fig. 6. Morton order for testcase2

Like in the previous test case, we divided our results into two categories based on message size - short and long. This test case is different from the previous example where the object was placed at the geometric midpoint of the domain. Here, majority of the points are distributed in the volume of the object which is entirely located in the left half of the domain at the start of the simulation. There is dense clustering of points in this region and this is not captured by Morton and the midpoint splitter of GenSFC. The maximum degrees of both Morton and GenSFC+midpoint is very high for this configuration, as can be seen from the T_{comm} values of short messages. The communication volumes are also high for these partitions due to high surface to volume ratio. Again, the experiments are designed to reflect weak scaling. The maximum number of blocks per processor per time step is almost the same across different values of $\#procs$. The results seem to be influenced largely by the asymmetry and clustering in the domain. For example, the highest values for T_{comm} are obtained when the domain has maximum asymmetry; it is a long narrow box with clustering in one quarter (32X4X4, 32X8X8, 64X16X8). Splitting along the geometric midpoint doesn't help in these cases.

We obtained the best results when the splitter was chosen to be a combination of median and midpoint. We split the domain along the median of the dimension of maximum spread at the top levels of the recursion, followed by midpoint at the lower levels. This created partitions with much better surface to volume ratio and reduced the maximum degree of the

#procs	init_domain	Short Msgs				Long Msgs			
		T_{comp}	T_{comm}	T_{lb}	T_{exec}	T_{comp}	T_{comm}	T_{lb}	T_{exec}
512	32X4X4	2.92	6.72	0.1323	15.64	22.77	22.29	0.11	51.02
1024	32X8X4	2.67	6.19	0.17	14.73	21.21	21.05	0.286	48.42
2048	32X8X8	2.67	6.81	0.393	16.98	19.80	19.29	0.44	45.84
4096	32X16X8	2.02	4.60	0.746	13.42	19.21	19.7	1.03	44.86
8192	64X16X8	2.00	4.93	1.84	15.20	21.00	22.03	2.06	49.457

TABLE VI
TOTAL RUNNING TIMES FOR TESTCASE2 WITH MORTON PARTITIONS FOR DIFFERENT PROCESSOR COUNTS

#procs	init_domain	Short Msgs				Long Msgs			
		T_{comp}	T_{comm}	T_{lb}	T_{exec}	T_{comp}	T_{comm}	T_{lb}	T_{exec}
512	32X4X4	2.92	6.62	0.14	15.40	22.78	21.98	0.19	50.92
1024	32X8X4	2.67	6.12	0.17	14.53	21.21	20.75	0.41	48.71
2048	32X8X8	2.67	6.01	0.40	16.04	19.80	19.26	0.51	46.02
4096	32X16X8	2.02	4.23	0.75	13.56	19.21	19.15	1.27	45.71
8192	64X16X8	2.00	4.82	1.91	15.36	21.00	21.93	2.32	49.37

TABLE VII
TOTAL RUNNING TIMES FOR TESTCASE2 WITH GENSF (MIDPOINT) PARTITIONS FOR DIFFERENT PROCESSOR COUNTS

#procs	init_domain	Short Msgs				Long Msgs			
		T_{comp}	T_{comm}	T_{lb}	T_{exec}	T_{comp}	T_{comm}	T_{lb}	T_{exec}
512	32X4X4	2.92	5.87	0.136	15.17	22.78	18.93	0.26	50.17
1024	32X8X4	2.67	5.68	0.19	14.91	21.21	18.63	0.62	49.24
2048	32X8X8	2.67	5.61	0.44	16.38	19.80	17.85	0.58	45.98
4096	32X16X8	2.02	3.70	0.76	13.43	19.21	17.34	1.41	44.91
8192	64X16X8	2.00	3.92	1.98	15.20	21.00	18.47	2.56	48.78

TABLE VIII
TOTAL RUNNING TIMES FOR TESTCASE2 WITH GENSF (MEDIAN) PARTITIONS FOR DIFFERENT PROCESSOR COUNTS

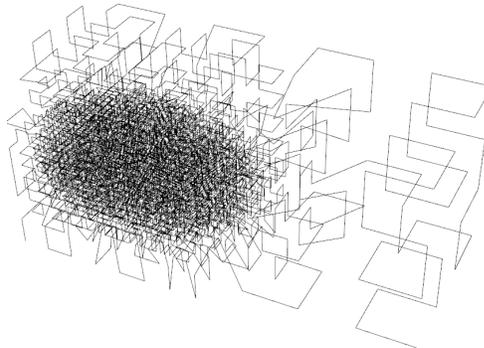


Fig. 7. GenSFC for testcase2

partitions drastically. The results are tabulated in table VIII. The worst case partitioning times are slightly higher than Morton, but as explained earlier, this is an upper bound. The values of T_{exec} continue to be affected by the inefficient tree data structure, so we don't see a big improvement in the total running time.

D. Testcase3

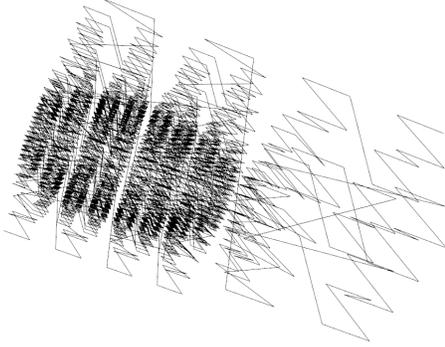


Fig. 8. Morton for testcase3

The final test case is similar to test case2, except that refinement was done only at the interface between the object and the domain. We used 5 levels of refinement and 2000 time steps. The object was moved at the same rate as the previous case. Figures 8 and 9 show the snapshots of space-filling curves traversing the domain during one of the early stages of the simulation. Note the dense refinement region along the boundary of the object. This was designed to resemble refinement along slowly moving fronts, commonly seen in scientific simulations. The refinement frequency, block size, stencil and number of variables are the same as the previous two test cases. Again, we tried to create experiments to show weak scaling. The shape of the domain (*init_domain*) was modified to introduce asymmetry.

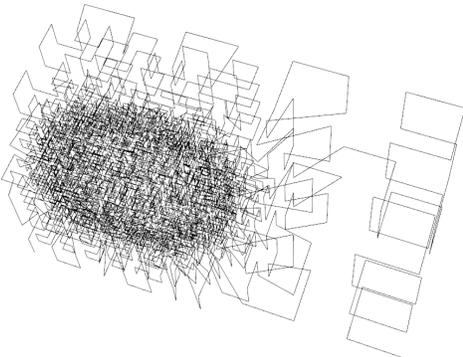


Fig. 9. GenSFC for testcase3

From our measurements, we found Morton and GenSFC+midpoint to have very high maximum degree values. This lead to an increase in their T_{comm} values for

#procs	init_domain	Short Msgs				Long Msgs			
		T_{comp}	T_{comm}	T_{lb}	T_{exec}	T_{comp}	T_{comm}	T_{lb}	T_{exec}
864	24X6X6	3.86	9.89	0.787	24.92	29.45	31.86	1.07	71.833
1024	32X8X4	4.39	10.67	1.129	27.14	33.29	36.15	1.09	79.725
1372	28X7X7	3.09	8.54	1.27	24.40	31.26	35.93	2.40	82.124
2048	32X8X8	2.70	8.36	2.33	28.09	26.18	29.36	2.72	72.08
4096	32X16X8	2.56	8.28	2.97	29.62	14.82	15.76	3.269	42.633
8192	64X16X8	1.95	5.65	5.22	25.57	20.21	23.15	5.49	59.57

TABLE IX

TOTAL RUNNING TIMES FOR TESTCASE3 WITH MORTON PARTITIONS FOR DIFFERENT PROCESSOR COUNTS

#procs	init_domain	Short Msgs				Long Msgs			
		T_{comp}	T_{comm}	T_{lb}	T_{exec}	T_{comp}	T_{comm}	T_{lb}	T_{exec}
864	24X6X6	3.86	9.90	0.75	24.89	29.45	31.12	1.15	72.02
1024	32X8X4	4.389	10.40	1.19	26.91	33.29	35.96	1.20	78.63
1372	28X7X7	3.09	7.94	1.16	22.68	31.26	35.45	2.49	80.61
2048	32X8X8	2.70	8.33	2.33	28.13	26.18	29.03	2.81	71.73
4096	32X16X8	2.56	8.12	3.03	29.94	14.82	15.23	3.8	41.26
8192	64X16X8	1.95	5.01	5.30	25.87	20.21	22.89	5.63	58.92

TABLE X

TOTAL RUNNING TIMES FOR TESTCASE3 WITH GENSF(MIDPOINT) PARTITIONS FOR DIFFERENT PROCESSOR COUNTS

#procs	init_domain	Short Msgs				Long Msgs			
		T_{comp}	T_{comm}	T_{lb}	T_{exec}	T_{comp}	T_{comm}	T_{lb}	T_{exec}
864	24X6X6	3.84	8.91	0.78	24.48	29.45	29.93	1.57	71.38
1024	32X8X4	4.39	9.13	1.19	25.87	33.29	33.74	1.41	76.18
1372	28X7X7	3.09	7.38	1.17	22.29	31.26	32.03	2.86	79.56
2048	32X8X8	2.70	7.42	2.33	27.68	26.18	27.12	3.23	72.74
4096	32X16X8	2.56	7.08	3.2	28.75	14.82	14.96	4.15	40.77
8192	64X16X8	1.95	3.89	5.54	25.13	20.21	21.18	5.80	57.63

TABLE XI

TOTAL RUNNING TIMES FOR TESTCASE3 WITH GENSF(MEDIAN) PARTITIONS FOR DIFFERENT PROCESSOR COUNTS

short messages. We were able to balance the degree of the partitions by splitting along the median of the distribution. As in test case2, the splitter shifted to midpoint of the dimension of maximum spread at the lower levels of recursion. There is significant lowering of communication times using the modified splitter. Therefore, this seems to be a good approach to identify clustering in arbitrary point distributions and obtain partitions with better surface to volume ratio. In all of the above testcases, we have shown a slice of the simulation where the distribution is evolving slowly.

VII. RELATED WORK

Parallel AMR is an area of active research and there are many frameworks available which have displayed scaling to large processor counts. GrACE [9] and Paramesh [7] are both frameworks for tree-based AMR implementation and both use SFC as partitioner of choice. Paramesh, which is used by FLASH [10], computes Morton order of the entire tree, including terminals and non-terminals. GrACE uses a Peano-Hilbert ordering of the domain at the top level and orders blocks according to this. They seem to have an efficient distributed tree data structure where the blocks are indexed using SFC keys. But it has not been optimized further; there is significant communication overhead to maintain this design. Both implementations have not analysed the quality of their SFC partitions in terms of degree and communication volume. Boxlib [11], Chombo [12] and SAMRAI [13] are software packages which use a patch-based implementation. They may further divide a patch into blocks. Boxlib stores and partitions

the the adaptive grids in the hierarchy according to their refinement levels. All the grids at a particular level are partitioned across all processes independently. This can adversely affect communication, since locality between levels is lost. Chombo is based on Boxlib, however, they partition the *disjoint boxes* that comprise an adaptive mesh using Morton order. The boxes may be of different sizes. They use a knapsack algorithm to load balance these boxes across processes. The quality of these partitions in terms of load balance and communication cost have not been evaluated. Also, this problem is constrained by load balancing at the granularity of boxes instead of smaller blocks. Besides, the meta data containing the grid hierarchy is maintained by all processes. Whenever the grids are modified, this meta data information needs to be updated. SAMRAI uses SFC-like ordering to arrange patches of different sizes in a linear list and balances the load by distributing the list. Enzo [14] is an open-source AMR package that has a patch-based implementation. It support blocks of arbitrary sizes. It partitions the top level grid using an SFC order, but the lower levels are kept local to a process. This can lead to load imbalance. They overcome this to some extent by having each over-loaded process occasionally share its load with other processes. The communication cost of this greedy load balancing scheme can be high if data is migrated to distant processes frequently. The effectiveness of this methods depends to a large extent on the amount of clustering in the mesh. There has been work in the algorithms community to define parallel graph partitioners that minimize edge-cut [15], [16] and [17]. They have high re-distribution costs. [18] describes a parallel algorithm based on graph partitioning that reduces re-distribution cost. [19] explored SFC partitions to some extent, but their parallel algorithm was not effective. New partitions were computed by exchanging additional workload with neighboring processes. [20] discusses an SFC-like mapping to generate quick partitions, but locality may be lost when new points are added or when they move.

VIII. CONCLUSION

Our SFC algorithm is able to identify clustering of points in the adaptive mesh and partition accordingly. This created partitions of better quality. The current version applies median splitting only on the initial data. The results reported in this paper are valid for a simulation window where the median doesnot shift by a large amount. If there is a large deviation from the current median value, partition quality will drop. We are currently working on implementing a parallel version of the median splitter so that this technique can be used adaptively whenever the domain changes. We are also working on improving the data structures used to represent the AMR grid hierarchy so that look-up and update operations on the tree can be done inexpensively without the need for global information at every process.

ACKNOWLEDGMENT

The work was supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

REFERENCES

- [1] M. J. Berger and J. Olinger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.
- [2] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.
- [3] H. Saagan, *Space-Filling Curves*. Springer-Verlag, 1994.
- [4] I. Beichl and F. Sullivan, "Interleave in peace, or interleave in pieces," *IEEE Comput. Sci. Eng.*, vol. 5, no. 2, pp. 92–96, Apr. 1998.
- [5] H. Sagan, "A three-dimensional hilbert curve," *International Journal of Mathematical Education in Science and Technology*, vol. 24, no. 4, pp. 541–545, 1993.
- [6] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.
- [7] P. MacNeice, K. M. Olson, C. Mobarri, R. de Fainchtein, and C. Packer, "Paramesh: A parallel adaptive mesh refinement community toolkit," *Computer Physics Communications*, vol. 126, no. 3, pp. 330 – 354, 2000. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0010465599005019>
- [8] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.
- [9] M. Parashar, James, and C. Browne, "Systems engineering for high performance computing software: The hdda/dagh infrastructure for implementation of parallel structured adaptive mesh refinement," in *In Structured Adaptive Mesh Refinement Grid Methods, IMA Volumes in Mathematics and its Applications*. Springer-Verlag, 1997, pp. 1–18.
- [10] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo, "Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes," *The Astrophysical Journal Supplement Series*, vol. 131, no. 1, p. 273, 2000. [Online]. Available: <http://stacks.iop.org/0067-0049/131/i=1/a=273>
- [11] "BoxLib," <https://ccse.lbl.gov/BoxLib/>, 2011.
- [12] P. Colella, D. T. Graves, J. N. Johnson, H. S. Johansen, N. D. Keen, T. J. Ligocki, D. F. Martin, P. W. Mccorquodale, D. Modiano, P. O. Schwartz, T. D. Sternberg, and B. V. Straalen, "Chombo software package for amr applications design document," LBNL, Tech. Rep., 2003.
- [13] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott, "Large scale parallel structured amr calculations using the samrai framework," in *In Supercomputing*, 2001.
- [14] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J. hoon Kim, M. Kuhlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C. So, F. Zhao, R. Cen, Y. Li, and T. E. Collaboration, "Enzo: An adaptive mesh refinement code for astrophysics," *The Astrophysical Journal Supplement Series*, vol. 211, no. 2, p. 19, 2014.
- [15] C. Walshaw, M. G. Everett, and M. Cross, "Parallel dynamic graph partitioning for adaptive unstructured meshes," *J. Parallel Distrib. Comput.*, vol. 47, no. 2, pp. 102–108, Dec. 1997. [Online]. Available: <http://dx.doi.org/10.1006/jpdc.1997.1407>
- [16] C.-W. Ou and S. Ranka, "Parallel incremental graph partitioning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 8, pp. 884–896, Aug. 1997. [Online]. Available: <http://dx.doi.org/10.1109/71.605773>
- [17] K. Schloegel, G. Karypis, and V. Kumar, "Wavefront diffusion and lmsr: Algorithms for dynamic repartitioning of adaptive meshes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, no. 5, pp. 451–466, May 2001. [Online]. Available: <http://dx.doi.org/10.1109/71.926167>
- [18] P. Diniz, S. Plimpton, B. Hendrickson, and R. Leland, "Parallel algorithms for dynamically partitioning unstructured grids," 1995.
- [19] J. R. Pilkington and S. B. Baden, "Dynamic partitioning of non-uniform structured workloads with spacefilling curves," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 288–300, 1995.
- [20] C.-W. Ou, S. Ranka, and G. Fox, "Fast and parallel mapping algorithms for irregular problems," *The Journal of Supercomputing*, vol. 10, no. 2, pp. 119–140, 1996. [Online]. Available: <http://dx.doi.org/10.1007/BF00130706>