MiniAMR - A miniapp for Adaptive Mesh Refinement

Aparna Sasidharan
University of Illinois at Urbana-Champaign
sasidha2@illinois.edu

Marc Snir
Argonne National LaboratoryChicago,IL
and University of Illinois at Urbana-Champaign
snir@illinois.edu

August 4, 2016

Abstract

We present a new implementation of MiniAMR, a miniapp for adaptive mesh refinement (AMR) that mimics commonly found workloads and communication patterns in AMR applications. We obtain significant performance improvements by using new data structures and algorithms. One of our important areas of focus in this work is the use of low-overhead distributed load balancing schemes for octree partitioning. We evaluate this new implementation by comparing its performance to that of the MiniAMR code in the Mantevo suite of miniapplications for three realistic test cases. In addition, the new code supports refinement in time (sub-cycling).

1 Introduction

Adaptive Mesh Refinement is a frequently used technique for efficiently solving partial differential equations (PDEs): A finer mesh is used in regions that require a higher resolution (e.g., because of turbulence), and a coarse mesh is used to cover less sensitive regions. This reduces both computation and storage, compared to a mesh with uniform resolution. Since the regions requiring a finer mesh may vary during the course of a simulation, the mesh is adaptively refined or coarsened to reflect these changes. This affects the amount of computation done by each processor; over time, it becomes necessary to re-partition the mesh across processors.

A good partition minimizes the amount of communication between processes and balances their computational load, thus reducing execution time. This has to be balanced against the overhead of computing the partition, migrating

data and re-constructing the distributed data structures that are used by the application. Many AMR frameworks use Space filling curves (SFC) for mesh partitioning, as they achieve a good compromise between these conflicting goals.

We focus in this paper on octree-based AMR [18, 25, 23, 19, 1, 21, 5]. Our paper makes the following contributions:

1. We present an improved linearized data structure for representing octrees. We also describe a low-overhead refinement and coarsening algorithm for octree based AMR.

2. We propose an amortized load balancing scheme that lowers the total execution time by reducing the frequency of re-partitioning.

3. We provide a low-overhead distributed weighted slicing algorithm for SFC that can be used for load balancing AMR applications which use sub-cycling.

These changes do not affect the numerical properties of the algorithms in any way, but only change their performance.

We evaluate the improvements by using the miniAMR benchmark in the Mantevo benchmark suite [10]. We compare the performance of the miniAMR-code in Mantevo to our new code, for three realistic test cases, using the same SFC (Morton order) for both. With the improved data structure and algorithms we could reduce the refinement time of MiniAMR by more than half. The computation time of the kernel also showed significant improvement by almost $1.8X$. Amortized load balancing was able to improve the running time of the simulation by a maximum of $3X$ for one of the test cases. We obtained significant performance improvements for the test cases with sub-cycling. The weighted slicing algorithm was able to reduce the execution time of the simulation by a maximum of $3X$. In addition to performance improvements, our code is more general as it provides support for sub-cycling.

The remainder of the report is organized as follows:

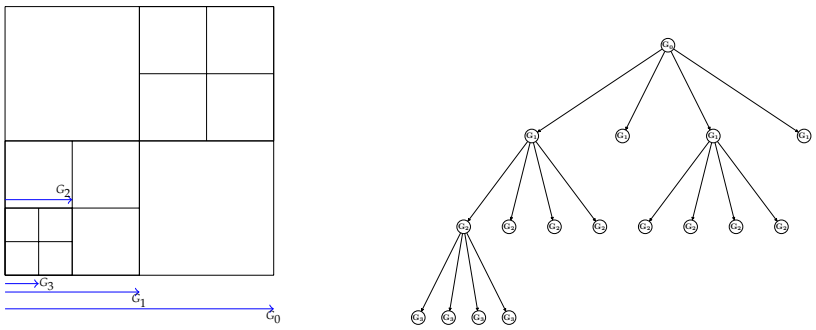
We discuss octree-based AMR in the next section; it also includes a short explanation of parallel octree partitioning using Morton order. Section 2 describes the performance model that we used for measuring execution time. Section 3 describes in detail the improvements we made to the Mantevo miniAMR code. Section 4 contains our experimental results. We finish with a short survey of related work in Section 5 and a brief conclusion.

1.1 Adaptive Mesh Refinement (AMR)

While many of our techniques generalize to other mesh-based algorithms, we shall focus in this work on octree-based AMR. Results for other AMR variants will be quantitatively different but we expect them to show improvements as well.

The miniAMR mini-app from the Mantevo suite is a compact proxy for octree-based AMR. It exhibits computation and communication patterns that

are typical of the application while being shorn of many of the details of a full application. MiniAMR uses 3D meshes with an octree structure: contiguous mesh cells are grouped into cubic *blocks*, the granularity of which can be decided by the user. Each block has a *halo* region defined around its mesh cells, which is used for exchanging boundary information with neighboring blocks. If a block needs to be refined, then it is split into 8 cubic sub-blocks; if blocks are coarsened, then the 8 adjacent sub-blocks are replaced by the original parent block. The refinement preserves a 2:1 *balance* condition: The refinement levels of adjacent blocks differ by at most one. The mini-app uses a 7-point stencil so that any two blocks are neighbors if they share a face. Computation typically involves only the leaves of the octree. Our mini-app also models applications which perform sub-cycling.



(a) Adaptively Refined Mesh (b) Octree

Figure 1

On an abstract level, the mesh data structure is a dynamic forest of octrees with the roots of the forest being the initial coarse blocks of the mesh. In our discussion, octree nodes are equivalent to the mesh blocks defined earlier. We will be using these terms interchangeably. This data structure should support the following operations:

- Refinement: 8 nodes are created as children of a previous leaf node.
- Coarsening: 8 sibling nodes are deleted and their parent becomes a leaf node.

Figures 1a and 1b provide an example for an adaptively refined mesh that satisfies the 2:1 balance criterion and its corresponding quad-tree in 2D.

For a parallel computation, the octree data structure is distributed across the compute nodes. The distribution should balance the amount of computation performed by each node, while reducing communication. The major communication overheads in octree-based AMR are the following:

- Balancing: The refinement and coarsening operations have to preserve the 2:1 balance condition. The refinement of a node could trigger further

refinement of its coarse neighbors. Similarly, the coarsening of a group of nodes could enable the coarsening of a neighboring group. Both operations require communication between a node and its neighbors.

- Updating block halo regions: Blocks exchange boundary data with their neighbors in every timestep of the simulation. The communication overhead involved depends on the type of finite difference scheme used. In the simplest case, the simulation is advanced only at the leaf nodes of the octree. In this case, only the leaf blocks perform boundary exchange. If the kernel performs refinement in time as well, then the rate of halo exchange between neighboring blocks depends on their relative refinement levels. The overheads are different for each kernel type. But all variants benefit from having a domain decomposition of the octree mesh that is load balanced and has good locality, i.e adjacent blocks are mostly stored on the same node.

Typically, a parallel octree meshing framework starts with an initial partition of octree nodes across processes and adaptively modifies it by refining and coarsening the tree nodes during the course of the simulation. This could result in partitions with load imbalance and worsened communication during balancing and halo exchange. Therefore, the octree needs to be re-partitioned periodically. This involves computing a new partition, migrating data, and re-establishing the information required for halo-exchanges. Space-filling curves are widely used for partitioning adaptively refined meshes. An SFC is used to create a linear ordered list of mesh blocks (tree leaves) which are then distributed among the participating processes in a load balanced manner. We also maintain the invariant that an internal block is stored on the same node as one of its children.

1.2 Octree Partitioning

SFCs can be used to partition rectilinear grids: each grid cell is represented by a point, usually located at its center of gravity. The curve is generated using the spatial co-ordinates of the points. A monotonic mapping function is used to convert the spatial co-ordinates into unique keys. The values of keys generated by the function should preserve the traversal order of the curve. The points are then partitioned into p subsets by partitioning the set of keys into p segments each containing roughly the same number of points, ensuring load balance. The partitions have good locality, so that, if the computation requires communication between adjacent grid cells, they will have low communication.

A *Discrete k-dimensional Space Filling Curve* (SFC) is a bijective mapping $C : \{1 \dots N^k\} \rightarrow \{1 \dots N\}^k$ such that $d(C(i), C(i+1)) = 1$ for all $0 < i < N^k$, where d is the Euclidean distance [9]. We define *good locality* by the requirement that any k consecutive points $C(i), \dots, C(i+k-1)$ on the SFC are contained in a hypercube of dimension $O(k^{1/m})$, where m is any non-zero integer. Some of the popular SFCs include Hilbert, Peano and Sierpinski [20]. Although Morton

order [2] does not fit into the above definition, it is widely used due to its ease of computation. The mapping function for Morton order is a simple bit-interleaving of the spatial co-ordinates of the points.

In this discussion, we have used Morton order for all of our experiments. We are currently working on replacing Morton order with SFCs having better locality in MiniAMR. For the adaptive octree mesh, we avoid re-computing the SFC from scratch whenever the octree is refined or coarsened. To do so, when a block with key ℓ is refined, it is replaced on the SFC by the eight children with keys ℓxyz labeled in the traversal order of the curve (Morton order in this case). When 8 sibling blocks are coarsened, then the coarse block (parent) replaces one of the children blocks on the SFC, and the other children are deleted. For Morton order, since the 8 siblings were contiguous on the SFC, they had binary keys ℓxyz and the coarse block is re-labeled ℓ; the lexicographic order of keys is preserved. Furthermore, each process can perform these updates independently in parallel, for those blocks it owns.

1.2.1 Sorting

In order to re-partition, we need to compute the (weighted) index of any point (leaf node) on the SFC. When a node is refined, its children are stored on the same process as their parent; when nodes are coarsened, the parent node already is stored on the same process as one of the children. Therefore, refinement and coarsening operations maintain the invariant that all nodes stored at process P_i have keys that are strictly less, in lexicographic order, than nodes stored at process P_{i+1}. As a result, it is sufficient to sort keys locally at each process, in order to obtain a global sort.

Each process locally numbers its keys in sorted order. A parallel prefix operation is then used to compute the global rank of the last key. After this step, processes can independently compute the global ranks of their local nodes and re-distribute them in a load balanced manner.

In the case where the mesh is refined in time, we treat the blocks as having a weight value proportional to their refinement level (since finer blocks are updated more often). The curve should then be sliced into equal weight segments.

2 Performance Metrics

In this section we discuss the performance model and the metrics we used to evaluate the performance of our mini-app. The AMR kernel consists of the following non-overlapping phases that are performed at different frequencies. The kernel iterates until convergence and we consider each iteration to be a single time step.

1. Refinement/Coarsening: The kernel checks for the refinement criteria every *ref_freq* time steps, and, if indicated, performs the refinement and coarsening of blocks. The addition and/or deletion of blocks is subject to the 2:1 balance criteria discussed earlier.

2. Load Balancing: Re-partitioning of nodes may be performed after refinement/coarsening. This includes computing the new partition, re-partitioning the data and computing the new communication pattern to be used in subsequent timesteps.

3. Computation: At every timestep each leaf node updates its data points based on neighbor values, as defined by a stencil pattern (7-point in this discussion). This step requires data exchange across the shared faces of leaf nodes (referred to as halo-update).

We measure, at each time step i and each process j, the time $T_{comp}(i,j)$ spent computing, time $T_{comm}(i,j)$ spent in halo-updates, time $T_{ref}(i,j)$ spent refining and coarsening the mesh, and time $T_{lb}(i,j)$ spent load balancing. Let n be the total number of timesteps, $ref\_freq = r$, and let $t_1, \ldots, t_k$ be the timesteps where load balancing occurs. Then, the total execution time, T_{exec}, can be defined as

$$T_{exec} = \sum_{t=1}^{n} T_{iter}(t) + \sum_{t=1}^{\frac{n}{r}} T_{ref}(rt) + \sum_{i=1}^{k} T_{lb}(t_i)$$

where,

$$\begin{aligned} T_{iter}(i) &= max_j(T_{comp}(i,j) + T_{comm}(i,j)), \\ T_{ref}(i) &= max_j T_{ref}(i,j), \text{ and} \\ T_{lb}(i) &= max_j T_{lb}(i,j). \end{aligned}$$

The equations presented here are for a general case without any computation-communication overlap in the kernel. However, our implementation uses a computation-communication overlap in the stencil code. If there is full overlap then the equation for T_{iter} becomes

$$T_{iter}(i) = max_j(max(T_{comp}(i,j), T_{comm}(i,j)))$$

We used these equations to compute the total execution time for all the test cases in this report.

3 MiniAMR

We have developed an improved implementation of the MiniAMR mini-app that is part of the Mantevo suite [10]. This mini-app was designed to better understand the communication overheads in an octree-based AMR, and use the insights to design better algorithms with lower communication costs. Therefore, the computation kernel is a simple 3D 7-point averaging stencil. It accepts an initial distribution of blocks and a refinement pattern in the domain. The refinement criteria (rate) and load balancing frequency are also inputs to the application. The blocks which overlap with the refinement pattern are marked

for refinement. The shape and area of refinement can be controlled to generate different test cases; it can be made to expand or move with a constant velocity in the domain. In the next subsection we discuss critical design decisions and optimizations which significantly improved the performance of MiniAMR over its counterpart in the Mantevo suite. Better partitions should have a more significant impact on communication time if a more complex stencil pattern is used. On the other hand, the use of a more compute-intensive kernel will reduce the relative weight of communication (but not its total weight).

We present in the following section optimizations we introduced in the miniAMR code and next section discusses their impacts. These improvements are specific to octree-based AMR.

3.1 Optimizations

3.1.1 SFC Key-indexed Data Structure

Distributed pointer-based data structures like trees, are typically implemented by explicitly maintaining their node locations, i.e remote process id and global node index. This requires additional book-keeping during updates to the data structures; refinement and coarsening in this case: After every refinement and balancing phase, the neighborhood information is updated by sending and receiving messages containing the global ids of new blocks. Also, one needs additional communication after load balancing to update the new locations of neighbors.

The use of SFC keys instead of pointers reduces the overhead of these operations, since the keys provide a level of abstraction that is independent of the underlying data decomposition. The only information required at every process is a mapping function/table that can derive the location of a block given its SFC key. This idea was explored in [16], but they stored the entire block dataset in a hashtable which affected memory locality and performance. In our implementation, each process maintains a local two-level data structure. The first level is a dictionary of its local SFC keys and the second level is the contiguous blocks array in memory. Look-up is performed using the dictionary that converts an SFC key into a location in memory. We have implemented the dictionary as a sorted array and for all of our test cases, it was small enough to fit entirely in cache, so that search for a key is very fast. An alternative implementation would be to use a hash table. This data structure reduces the work required to compute the new communication pattern after data migration during a load balancing phase. The only information required at each process is the key range (minimum and maximum keys) of the other partitions. This information is exchanged using a single all-to-all communication after data migration. For all the test cases considered in this report the non-terminal nodes of the octree do not perform any computation, therefore, we do not store them.

3.1.2 Refinement-Coarsening Algorithm

The algorithm has two phases: the *consensus* phase and the *refinement-coarsening* phase. This algorithm is executed during every refinement phase of the kernel.

Consensus Phase: This phase of the algorithm is iterative and is repeated until quiescence (termination). We define what we mean by quiescence later.

Blocks are initially marked for refinement based on whether or not their centers coincide with the current refinement region. The blocks that are not marked for refinement or are forced to stay at their current level to satisfy balance constraints are assumed to coarsen since their refinement levels are higher than the desired level. Therefore, a block can be in one of three states - *refine*,*stay* or *coarsen*. The default state of a block is assumed to be *coarsen*. Any change in the state of a block will need to be communicated to its neighbors and siblings. Each block stores the current states of its neighbors and updates this information as the algorithm executes. This communication is highly localized and can be executed concurrently for different regions of the mesh. In other words, the *region of influence* of a refining or coarsening block is limited. We make two assumptions here, that the mesh is balanced before entering a refinement-coarsening phase and that the refinement level of a block can change by at most one during a refinement phase. Our algorithm is similar to the prioritized ripple propagation algorithm in [25], but we have lower communication overheads due to the following :

- Neighbors are informed only during a change of state
- There is no synchronization between levels

Besides, the algorithm in [25] does not perform coarsening of leaf blocks.

Each process maintains a local queue of blocks marked for change. A single iteration of the consensus algorithm performs the following steps :

1. Process entries in the local queue until it is empty.
2. Update the states of neighbors and siblings that are local (on the same process)
3. Aggregate messages to non-local neighbors and siblings in message queues. We maintain a message queue for each neighbor in the communication graph.

The messages aggregated by a process during an iteration are exchanged using the Sparse Collective routines [11] in MPI. This exchange of states could trigger the refinement of more blocks at every process. The new blocks marked for refinement are added to the local queue for processing in the next iteration. The consensus algorithm terminates when the local queues and message queues on every process are empty. We define this state as quiescence; when all the processes in the system are idle. We check for quiescence at the end of every iteration of the consensus algorithm. This is the only synchronization point in

this algorithm. Each process exchanges a short integer indicating whether it is idle (its local queue and message queues are empty) or not. This integer value is reduced to check if all the processes are idle. This is done using a single $MPI\_Allreduce$ function call. A pseudo-code for the algorithm is provided below (Algorithm 1).

Algorithm 1 Parallel Consensus Algorithm

```
1:  procedure ParallelConsensus
2:      while ¬Terminate do
3:          while ¬q.empty() do
4:              n = q.pop()
5:              nbrs = blocks[n].nbrlist()
6:              sibs = blocks[n].slist()
7:              for all nbr ∈ nbrs do
8:                  if nbr.is_local() then
9:                      local_update(n, nbr)
10:                 else
11:                     aggregate_msg(nbr)
12:                 end if
13:             end for
14:             for all sib ∈ sibs do
15:                 if sib.is_local() then
16:                     local_update(n, sib)
17:                 else
18:                     aggregate_msg(sib)
19:                 end if
20:             end for
21:         end while
22:         MPI_Neighbor_alltoallw()
23:         MPI_alltoall(terminate)
24:     end while
25: end procedure
```

There are non-iterative versions of the parallel-consensus algorithm [4], [21]. However, in practice we found our algorithm converges quickly since refinement and coarsening are localized operations. Also, since processes communicate only with their nearest neighbors (same communication pattern as halo-update), the number of messages and message sizes per process are very small. The only information a block sends to its neighbor is its state (one integer). The non-iterative version on the other hand has fewer iterations, but the number of messages and message sizes per process can be quite high. We intend to perform a more detailed comparison between our algorithm and the non-iterative algorithm in future.

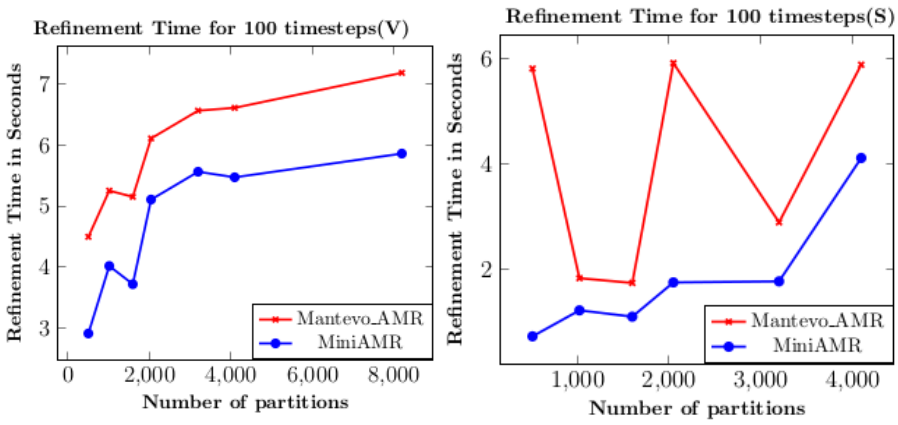


Figure 2: Refinement Time on Vesta and Stampede

Refinement-Coarsening Phase: This is the phase where the addition and deletion of blocks takes place along with an update of the neighborhood information. New blocks are added locally and blocks that are marked for coarsening delete themselves. Since each block is aware of the final decision of its neighbor blocks, the SFC keys of the neighbors can be updated locally by manipulating their current keys (adding or deleting bits). There is no communication in this stage.

3.1.3 Stencil

We implemented the stencil phase using complete overlap of computation and communication to reduce execution time. The messages for halo-updates are aggregated (packed) and sent. While waiting for halo-updates from remote neighbors, the processes perform local halo-updates and stencil computation within the interior of the block. The remote halo regions are updated when messages arrive. This is beneficial for AMR when the blocks have good volume to surface ratio, so that communication is fully covered by local computation.

3.2 Results

In this section we present a detailed comparison of our optimized AMR implementation against the equivalent mini-app from Mantevo, which will be referred to as Mantevo_AMR for the rest of this discussion.

Mantevo_AMR uses a regular pointer-based data structure for its AMR tree with explicit neighbor updates after refinement and load balancing. For a fair comparison, we used Morton ordering to enumerate and partition the blocks in both versions. Also, we triggered load balancing after every refinement, so that the executions of both programs exhibit the same behavior at all time steps.

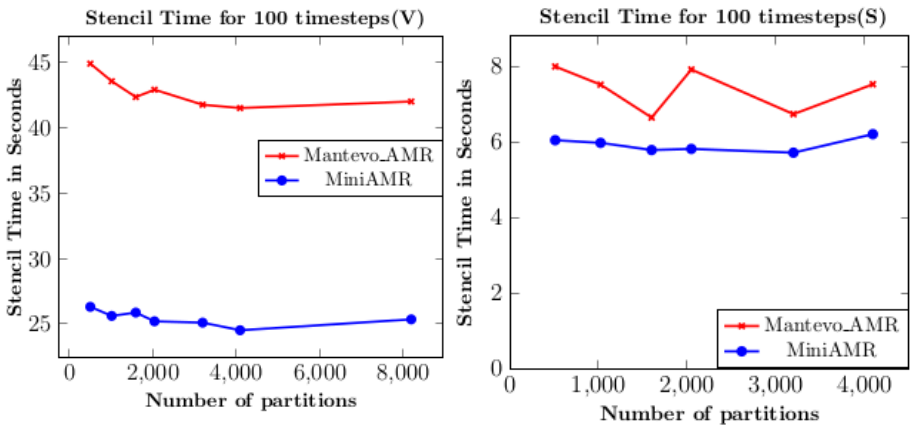


Figure 3: Stencil Time on Vesta and Stampede

The differences between the two versions is therefore in the tree representation, refinement algorithm and stencil phase. Also, Mantevo_AMR does not aggregate messages. Both codes are MPI only, with a process per core, and each process owning one partition of the mesh.

The test case used was that of an immersed moving sphere with refinement along its surface (where it interacts with the domain). The frequency of checks for refinement was set to 3 time steps, although addition and deletion of blocks took place only every $6 - 9$ timesteps.

Our experiments were carried out on Vesta which is a BG/Q machine at Argonne National Laboratory and Stampede, which is a supercomputer built from Intel SandyBridge NUMA processors, at the University of Texas, Austin. We used the version of MPI that has optimized on-node communication (MPICH+Nemesis on Vesta and MVAPICH+Nemesis on Stampede) [3]. All experiments in this section show weak scaling results where the number of blocks per process was kept roughly the same (approximately 300 blocks per process). Each block had $4 \times 4 \times 4$ grid cells. The compute nodes and interconnection networks in these machines differ greatly. Hence, the comparisons are relevant and give us valuable insights into useful program optimizations for each machine.

The graphs in figure 2 show the total refinement time for 100 time steps on both machines. V stands for Vesta and S for Stampede. Mantevo_AMR does refinement in stages, starting from the lowest refinement level to the highest. Refinement decisions are taken at the parent nodes instead of the leaf blocks themselves which results in additional communication between terminal and non-terminal blocks of the tree during refinement iterations. Besides, the locations of non-terminal blocks in the AMR tree are not updated during load balancing. So, they could potentially lie on any process irrespective of SFC order. This leads to a lot of communication overhead for Mantevo_AMR which worsens with increasing number of partitions. In addition to replacing commu-

nication for neighbor updates with local computation, our refinement algorithm also avoids communication with non-terminal nodes completely. Our refinement times are lower on both machines, probably due to these reasons. Vesta has slower processors with higher memory access times than Stampede. Therefore, we believe that the predominant cost on Vesta is due to memory accesses and cache misses. Off-node communication is a relatively significant bottleneck on Stampede due to its faster processors and slower network. This is evident in the graph in figure 2, where the refinement times for Mantevo_AMR on Stampede vary drastically, due to its irregular tree decomposition (non-terminals vs terminals) and inefficient algorithm.

The next set of graphs in figure 3 compare the time taken for stencil computation and communication for both miniapps on Vesta and Stampede. In Mantevo_AMR, each leaf block stores pointers to its data instead of the data itself which leads to pointer chasing and loss of memory locality. This affected its performance greatly on Vesta. Our results are better on Vesta; this seems to be due to the computation-communication overlap and contiguous layout of blocks in memory. On Stampede, we used the NUMA aware memory allocation policy to allocate blocks close to the sockets on which the corresponding processes were mapped. Once again, the performance of Mantevo_AMR suffers due to its inefficient communication routines and data structures. The stencil communication in Mantevo_AMR does not use full message aggregation, which meant there could be multiple messages between any two processes during a single communication step. This irregularity in communication shows up in its stencil time on Stampede. The stencil time for our mini-app on Stampede shows a slight increase at 4096 processes, which we believe is due to the discontinuities in Morton ordering which increases the off-node communication time.

4 Test Cases

This section explores additional test cases and techniques to lower the total execution time of an AMR simulation. We have used Morton order as the SFC of choice along with the performance model described earlier, i.e refinement time, stencil time and load balancing overheads for MiniAMR. All of our test cases use a simple computation kernel that performs a 7-point stencil operation. We used blocks with dimensions $4 \times 4 \times 4$, i.e, 4 grid cells in each dimension, maximum of 6 refinement levels and a single halo layer. All the graphs presented in this section show strong scaling results with total execution time over 800 time steps vs number of partitions. We measure total execution time as the sum of the time spent in refinement, stencil computation, halo-exchange and load balancing, following our performance model explained in section 2.

4.1 Test Case 1 – Hierarchical Partitioning

The first test case we considered is that of an expanding sphere, which closely mimics an explosion. Blocks are refined along the boundary of the expanding

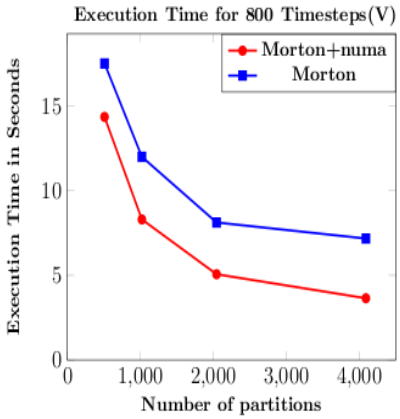


Figure 4: Test case 1 on Stampede for Morton with and without NUMA awareness

front. We checked for refinement/coarsening at even time steps ($ref\_req = 2$). The rate of expansion was based on measurements from real applications; there were $8 - 10$ time steps between addition/deletion of blocks. During expansion, blocks are added along the entire circumference of the sphere and removed from its interior. This leads to a uniform change in load along the SFC. We used this test case to study the impact of data locality on the total execution time. Since the nodes on Stampede are NUMA aware, we conducted the same experiment with and without numa-aware data placement. Morton order is naturally hierarchical - all subcells generated from a parent cell are traversed before visiting a neighbor of the parent. Therefore, when pieces of the curve are assigned to nodes, the partitions within a node are guaranteed to be connected by the SFC. What matters is the order of assignment of partitions within a node to individual cores. Since each node on Stampede has 16 cores, we assigned 16 MPI processes to it;one process per core. Each node on Stampede has two sockets;each socket having 8 cores. As expected, the best performance was obtained when the data resided on the same socket as the MPI process using it. We compared two different assignments : round-robin and contiguous. Round-robin alternates the assignment of partitions between the two sockets whereas in contiguous assignment, all the cores on a single socket are assigned before switching to the next socket. The results are presented in figure 4. This test case had an initial coarse mesh of $64X64X64$ blocks and simulated close to 200000 blocks.

The graph in figure 4 clearly shows the advantage of a contiguous assignment. It has better locality that matches the hierarchical nature of the SFC partitions. We used contiguous assignment for the remaining test cases in this report.

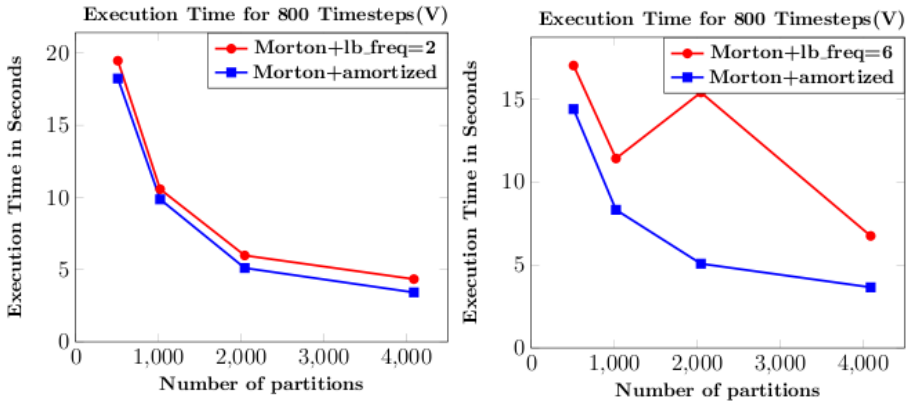


Figure 5: Test cases a and b for amortized load balancing

4.2 Test case 2 – Amortization of Load Balancing

This subsection introduces amortized load balancing to AMR. One of the contributing factors to the total execution time is the number of load balancing steps taken during the course of the simulation. We attempt to minimize the frequency of load balancing by amortizing the cost of a load balancing phase (including data migration) over the subsequent computation and communication steps. In the extreme case, one could trigger load balancing after every refinement/coarsening phase. This strategy will create good partitions at the cost of frequent re-partitioning overheads. In order to balance the two competing needs, we introduce the idea of *Amortization* [24] to load balancing. Formally, we split the computation into segments of ref_reg iterations, with a refinement check at the end of each segment; assume that load balancing is executed at the end of segment t_0; let C_{lb} denote the load balancing cost and let $T(t)$ be the time taken for the execution of segment t. The next load balancing phase will be at the end of segment t_1 where t_1 is the first time that

$$C_{lb} \leq \sum_{t_0+1}^{t} (T_t - T_{t_0+1})$$

This equation essentially captures by how much the current partition deviates from the *good* partition that was obtained immediately after load balancing, and when the difference is large enough, it automatically triggers load balancing. Amortization lowered the execution time of the simulations considerably.

The graphs in figure 5 show the execution times obtained for a simulation with 800 time steps on Stampede.

Both test cases had an initial domain covered by $64 \times 64 \times 64$ blocks and created a mesh with 220000 blocks (14080000 points). The explosion started as a point located at the center of this domain. For our measurements, we skipped the first 100 time steps to allow the simulation to attain a reasonable size (*steady*

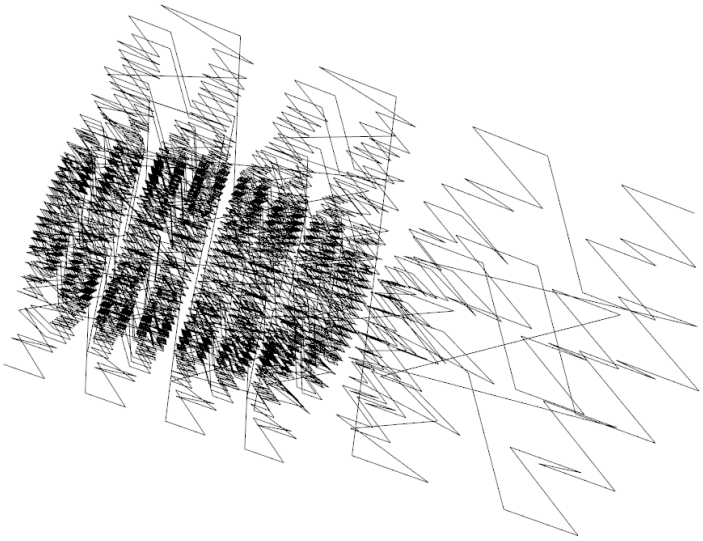


Figure 6: Snapshot of the adaptive mesh generated for Test Case 3

state). Both experiments use Morton ordering of blocks. The base case performs load balancing once every 6 time steps (matches with the rate of refinement). The difference between test cases a and b is in the rate at which the explosion front expands. This affects the rate at which computational load changes in the domain. The rate of expansion for test case a is much slower than that of test case b. In the case of test case b, frequent load balancing resulted in higher data migration to balance the load. Therefore, we obtained much better results with amortized load balancing for test case b. To benefit from amortization, the re-partitioning phase should be cheap compared to the stencil and refinement phases. This will allow the amortization equation to quickly detect variations in load across the partitions and trigger the next load balancing phase. We could achieve both goals without introducing additional overheads.

4.3 Test Case 3 – Weighted Partitions

This test case uses refinement in time and space dimensions. We use the term *refinement ratio* to define the ratio between the refinement levels (space or time) of adjacent blocks in the mesh. For this test case, both space and time dimensions have the same refinement ratio of 2. When an adaptive mesh does sub-cycling, the number of time steps executed by a mesh cell (block) is proportional to its refinement level. Therefore, the computational load is no longer uniform through out the mesh, which affects the load balance of the partitions. We assigned a non-zero weight to each block to capture its computational load.

The weight of a block, w, is computed as

$$w = r^l$$

where r is the refinement ratio of the mesh and l is the refinement level of the block.

Algorithm 2 Parallel Weighted Slicing Algorithm

```
1:  procedure PARALLEL W-SLICING
2:      my_load = sum_local_weights()
3:      total_load = Allreduce(my_load)
4:      load_per_proc = total_load/num_procs
5:      left = 0
6:      left = Ex_Scan(lmy_load)
7:      c_in = 0
8:      cur_wt = left
9:      cur_bin = 1
10:     while c_in < blocks.size() do
11:         my_wt = weight_of_block(c_in)
12:         if cur_wt + my_wt <= cur_bin * load_per_proc then
13:             assign_pid(c_in) = cur_bin − 1
14:             c_in = c_in + 1
15:         else
16:             cur_bin = cur_bin + 1
17:         end if
18:     end while
19: end procedure
```

We then slice the space-filling curve (Morton) into equal weight pieces. The pseudo code for our slicing algorithm is provided in (Algorithm 2). This is a distributed algorithm, with low-overheads and suitable for use with our amortized load balancing scheme. *my_load* is the sum of weights of all blocks owned locally by any process. *Ex_Scan* is an exclusive scan operation performed on the value of *my_load*.

We used a different scenario for this test case. This experiment models a spherical object of fixed size moving in a fluid domain and refinement is done along the boundary of the object with the domain. The simulation started with an initial mesh of $64X64X64$ blocks and reached a steady state size of approximately 200000 blocks (1280000 points). The graph in figure 7 compares the execution time obtained for weighted slicing of the curve against unweighted slicing. As expected, weighted slicing gives much better load balance for an AMR simulation that does sub-cycling. This experiment does load balancing at fixed intervals, i.e every 2 time steps. We also compared the performance of this test case with amortized load balancing.

Although amortization reduced the total execution time for the unweighted curve slicing by a small amount, it still performed poorly compared to the

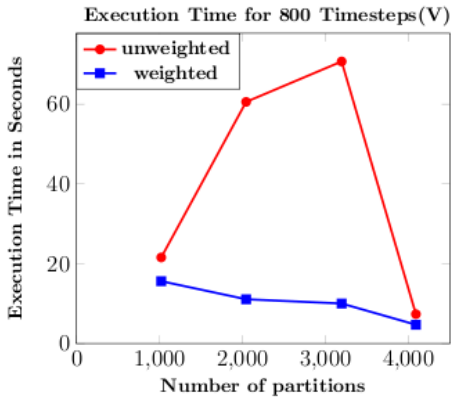


Figure 7: Test case 3 : Execution Time for weighted vs unweighted slicing

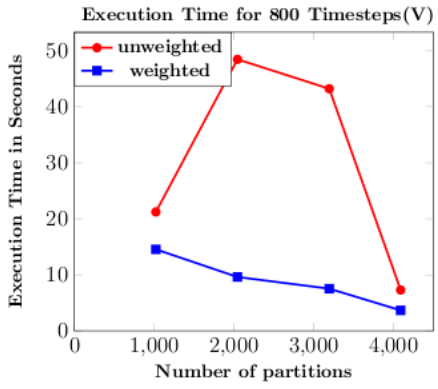


Figure 8: Test case 3 : Execution Time for weighted vs unweighted slicing with Amortized load balancing

weighted curve slicing. The results are presented in the graph in figure 8.

5 Related Work

Octree-based AMR is an area of active research and there are many frameworks available which have displayed scaling to large process counts. LibMesh [12] and deal.II [1] are libraries for finite element mesh generation that use adaptive refinement. But both libraries replicate the entire mesh information across all processes which affects their scalability. They use graph partitioners like Metis [22] to partition the mesh. GrACE [16] and Paramesh [13] are frameworks for octree-based AMR that use space filling curves for partitioning. Paramesh has an octree implementation that uses pointers and Morton ordering on the leaf nodes. The locations of neighboring octants are maintained explicity using their global ids and process ids. The refinement algorithm used in Paramesh is synchronous and does not aggregate messages sent and received during refinement. GrACE uses a Peano-Hilbert ordering of the domain and the octree is implemented as a hash table indexed using SFC keys. Although their octree does not use pointers, since the entire block data is stored in the hash table, it is large and therefore expensive to maintain. Octor [25] has a pointer-based octree data structure that is partitioned using Morton order and proposes a prioritised ripple propagation algorithm for 2:1 balancing. Their implementation is communication intensive since all blocks (irrespective of their refinement criteria) perform a neighbor search. The algorithm is level-synchronous which could potentially increase the number of iterations until consensus is achieved. P4est [4] is a software library for forest of octrees mesh generation that uses Morton ordering on the leaf nodes. Their data structures are similar to our implementation, however the refinement algorithm is considerably different. They use a non-iterative scheme for refinement instead of the ripple algorithm. The implementation by default does not allow coarsening of blocks distributed across processes, which is a slight deviation from the problem we address in this work. Dendro [21] has schemes for bottom-up construction of the octree mesh and a global coarsening algorithm in addition to non-iterative refinement. They use linear octrees constructed from Morton ordering of the leaf nodes. [6] has explored the use of Hilbert curves for partitioning octrees. They have shown the lower surface area of hilbert partiitons compared to Morton order. However, their implementation will work only for regular 2D and 3D meshes. The curve will have discontinuities if used on an unstructured mesh. Also, the hilbert keys used are not location codes which results in explicit communication to update neighbors during refinement as well as re-partitioning. [8] provides a good overview of some of the openly available state-of-the-art AMR libraries and codes.

There has been work in the algorithms community to define parallel graph partitioners that minimize edge-cut [26, 14, 22]. They have high re-distribution costs. [7] describes a parallel algorithm based on graph partitioning that reduces re-distribution cost. [17] explored SFC partitions to some extent, but

their parallel algorithm was not effective. New partitions were computed by exchanging additional workload with neighboring processes. [15] discusses an SFC-like mapping to generate quick partitions, but locality may be lost when new points are added or when they move.

6 Conclusion

This report presents a new version of the miniAMR benchmark for adaptive mesh refinement that can be used to study new partitioning schemes and load balancing strategies for AMR. This version is more general, as it supports subcycling. Being more optimized, it is representative of well-tuned AMR codes. We also describe optimizations to improve the performance of Octree based AMR - two-level SFC key indexed data structure, improved algorithm for mesh refinement and coarsening and amortized distributed load balancing. We have presented three different test cases for Octree-based AMR with varying load distributions to illustrate the advantages of the above optimizations. The load balancing strategies presented in this report, seem to have significantly improved the execution times of the simulations for all test cases. Further work is needed to separate the contributions of the various improvements, and to push additional optimizations, such as the use of a hybrid shared-memory/distributed-memory programming model. We are also working on using similar techniques for unstructured adaptively refined meshes.

7 Acknowledgements

We would like to thank the developers of the Mantevo suite of mini-apps at Sandia National Laboratory for their guidance at various stages of development of MiniAMR. They were generous in sharing their initial test cases and results. We would also like to thank the developers of MPICH library at Argonne National Laboratory and the technical support at TACC for promptly answering questions related to machine architecture and performance. We express our gratitude towards Dr. Anshu Dubey for her valuable guidance throughout the project and also while preparing this technical report.

References

[1] W. Bangerth, R. Hartmann, and G. Kanschat. Deal.ii— A General-purpose Object-oriented Finite Element Library. *ACM Trans. Math. Softw.*, 33(4), Aug. 2007.

[2] I. Beichl and F. Sullivan. Interleave in Peace, or Interleave in Pieces. *IEEE Comput. Sci. Eng.*, 5(2):92–96, Apr. 1998.

[3] D. Buntinas, G. Mercier, and W. Gropp. Design and Evaluation of Nemesis, a Scalable, Low-latency, Message-passing Communication Subsystem.

In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid2006)*, pages 521–530, May 2006.

[4] C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[5] J. J. Camata and A. L. G. A. Coutinho. Parallel Linear Octree Meshing with Immersed Surfaces. In *22st International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2010, Petropolis, Brazil, October 27-30, 2010*, pages 151–158, 2010.

[6] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco. Dynamic Octree Load Balancing using Space-filling Curves. Technical Report CS-03-01, Williams College Department of Computer Science, 2003.

[7] P. Diniz, S. Plimpton, B. Hendrickson, and R. Leland. Parallel Algorithms for Dynamically Partitioning Unstructured Grids, 1995.

[8] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, B. O'Shea, E. Schnetter, B. Van Straalen, and K. Weide. A Survey of High Level Frameworks in Block-structured Adaptive Mesh Refinement Packages. *J. Parallel Distrib. Comput.*, 74(12):3217–3227, Dec. 2014.

[9] C. Gotsman and M. Lindenbaum. On the Metric Properties of Discrete Space-filling Curves. *IEEE Transactions on Image Processing*, 5(5):794–797, May 1996.

[10] 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. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[11] T. Hoefler and J. L. Traff. Sparse Collective Operations for MPI. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[12] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. libmesh: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Eng. with Comput.*, 22(3):237–254, Dec. 2006.

[13] P. MacNeice, K. M. Olson, C. Mobarry, R. de Fainchtein, and C. Packer. PARAMESH: A Parallel Adaptive Mesh Refinement Community Toolkit. *Computer Physics Communications*, 126(3):330 – 354, 2000.

[14] C.-W. Ou and S. Ranka. Parallel incremental graph partitioning. *IEEE Trans. Parallel Distrib. Syst.*, 8(8):884–896, Aug. 1997.

[15] C.-W. Ou, S. Ranka, and G. Fox. Fast and Parallel Mapping Algorithms for Irregular Problems. *The Journal of Supercomputing*, 10(2):119–140, 1996.

[16] 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*, pages 1–18. Springer-Verlag, 1997.

[17] J. R. Pilkington and S. B. Baden. Dynamic Partitioning of Non-Uniform Structured Workloads with Space filling Curves. *IEEE Transactions on Parallel and Distributed Systems*, 7:288–300, 1995.

[18] S. Popinet. Gerris: A Tree-based Adaptive Solver for the Incompressible Euler Equations in Complex Geometries. *J. Comput. Phys.*, 190(2):572–600, Sept. 2003.

[19] D. Rosenberg, A. Fournier, P. Fischer, and A. Pouquet. Geophysical Astrophysical Spectral-element Adaptive Refinement (GASpAR): Object-oriented H-adaptive Fluid Dynamics Simulation. *Journal of Computational Physics*, 215:59–80, June 2006.

[20] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[21] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros. Dendro: Parallel Algorithms for Multigrid and AMR methods on 2:1 Balanced Octrees. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 18:1–18:12, Piscataway, NJ, USA, 2008. IEEE Press.

[22] K. Schloegel, G. Karypis, and V. Kumar. Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes. *IEEE Trans. Parallel Distrib. Syst.*, 12(5):451–466, May 2001.

[23] J. R. Stewart and H. C. Edwards. A Framework Approach for Developing Parallel Adaptive Multiphysics Applications. *Finite Elem. Anal. Des.*, 40(12):1599–1617, July 2004.

[24] R. E. Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[25] O. D. R. Tu Tiankai and G. Omar. Scalable Parallel Octree Meshing for Terascale Applications. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 4–, Washington, DC, USA, 2005. IEEE Computer Society.

[26] C. Walshaw, M. G. Everett, and M. Cross. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108, Dec. 1997.