A DISTRIBUTED MULTI-THREADED DATA PARTITIONER WITH SPACE-FILLING CURVE ORDERS

BY

APARNA SASIDHARAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Doctoral Committee:

Professor Marc Snir
Professor Laxmikant Kale
Professor Paul Fischer
Dr. Anshu Dubey, Argonne National Laboratory

ABSTRACT

The problem discussed in this thesis is distributed data partitioning and data reordering on many-core architectures. We present extensive literature survey, with examples from various application domains - scientific computing, databases and large-scale graph processing. We propose a low-overhead partitioning framework based on geometry, that can be used to partition multi-dimensional data where the number of dimensions is ≥ 2. The partitioner linearly orders items with good spatial locality. Partial output is stored on each process in the communication group. Space-filling curves are used to permute data - Morton order is the default curve. For dimensions ≤ 3, we have options to generate Hilbert-like curves. Two metrics used to determine partitioning overheads are memory consumption and execution time, although these two factors are dependent on each other. The focus of this thesis is to reduce partitioning overheads as much as possible. We have described several optimizations to this end - incremental adjustments to partitions, careful dynamic memory management and using multi-threading and multi-processing to advantage. The quality of partitions is an important criteria for evaluating a partitioner. We have used graph partitioners as base-implementations against which our partitions are compared. The degree and edge-cuts of our partitions are comparable to graph partitions for regular grids. For irregular meshes, there is still room for improvement. No comparisons have been made for evaluating partitions of datasets without edges. We have deployed these partitions on two large applications - atmosphere simulation in 2D and adaptive mesh refinement in 3D. An adaptive mesh refinement benchmark was built to be part of the framework, which later became a testcase for evaluating partitions and load-balancing schemes. The performance of this benchmark is discussed in detail in the last chapter.

ACKNOWLEDGMENTS

I wish to thank my advisor Professor Snir for his help and guidance during the entire duration of my Ph.D. I also wish to thank the other students of his group, both past and present for their help in times of need. Besides government funding agencies I would like to thank the Department of Computer Science, for supporting my education for 7 years and also providing me with the opportunity to teach and interact with students. I would like to thank Professors Kale and Zilles and their students, both of whom have helped me over the years as teaching assistant. I would like to extend gratitude towards my family and friends, near and far, for being supportive. I would like to thank TACC and ANL for allowing access to their computing resources, as well as the department technical supprt staff for helping in times of trouble. I wish to acknowledge my neighbors, old and new for keeping it quiet, as well as the town of Urbana, the co-op and farmer's market for their food. Most importantly, the birds and animals of Urbana will be remembered, especially, my pet cat, who was with me at a table, while I wrote this thesis.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

SFC	Space-filling curves
AMR	Adaptive Mesh Refinement
2D	Two-dimensions
3D	Three-dimensions
NUMA	Non-uniform Memory Access
ANL	Argonne National Laboratory
TACC	Texas Advanced Computing Resources
KNL	Knight's Landing
MPI	Message Passing Interface

CHAPTER 1: INTRODUCTION

1.1 INTRODUCTION

Partitioning data is an essential step in the parallelization of a problem. The goal is to produce good quality partitions that are load balanced and have low communication between process/threads. Most partitioning software are based on graph partitioners that solve for the lowest communication volume or edge cut subject to load balance constraints. Some of them generate very good partitions for general graphs. Historically, the formulations of this problem and its solutions have focussed on generating good quality partitions, largely ignoring their overheads. There has been comparatively less effort on the performance evaluation of partitioners as stand-alone software and their implementations in the parallel space. One of the challenges in this domain is the partitioning of raw data which do not have adjacency relationships. Distributed partitioning of large datasets using thousands of threads on many-core architectures is a problem that requires scalable solutions.

We address in this thesis possible solutions to these problems. We propose a low-overhead parallel data partitioner which produces good quality partitions and data permutations using space-filling curves. These permutations can be used to assign points to coarse and fine partitions - with coarse partitions mapped to processes and finer partitions mapped to cores and threads. We allow incremental adjustments to partitions - addition of new points or deletion of existing points. We have included benchmarks from scientific computing for evaluating partitions. The results are encouraging; it is a further push towards a programming model with MPI+threads, for scalable solutions to big data problems.

Partitioning is the first and most important step that determines the total execution time of any parallel program. Much can be achieved by starting from well-organized data, both in terms of computation and communication time within a node as well as communication between nodes. The converse argument is also true - much is lost in terms of resources if a parallel program is based on poor quality partitions. Considerable effort is spent on improving the performance of parallel programs using auto-tuning [1], tiling [2], compiler optimizations, loop re-ordering [3] etc, all of which are derivatives of the same idea - graph paritioning. There is plenty of work character-

izing the quality of partitions, with definitions for good and bad partitions. We will be discussing some of the commonly used metrics and their significance later. For distributed data, the metrics are usually edge-cut or communication volume, which is measured as the number of point-to-point messages and the total number of bytes exchanged. There are more complex models that include machine parameters. The virtual topology of the machine with number of hops, routing overheads and congestion in the network are costs that can be added to the base costs. However, we have considered only the most expensive terms in the network model for evaluating our work. Partition metrics are chosen in relation to metrics in communication - i.e message latency and bandwidth. Some of these terms are dependent on each other in some sense. If a partitioner minimizes the maximum number of messages between any two processes in the network, it indirectly reduces congestion between nodes. The same can be argued about communication volume, lower edge-cut results in fewer packets to route. Mapping communicating neighbors to nearest nodes on the machine can make some difference in the communication time by reducing maximum number of hops. But, mapping functions and metrics depend on the machine topology.

At the second level of partitioning, within a node, distributed memory metrics are replaced by equivalents in shared memory. In a simple model, the edge-cut or communication volume can be replaced by number of cache misses. There are more detailed models involving caches and multiple NUMA nodes. Although we have not developed detailed models for shared-memory communication, we have re-ordered data accesses within a node to increase cache re-use. There are also testcases where we have mapped data to the closest NUMA nodes to reduce latency of memory accesses. These optimizations are hard to generalize because they are architecture dependent and need to be tuned for the processor used.

This work is based on geometric partitioning that ignores adjacencies, but reduces edge-cut by using a distance metric for separating data. The points co-located in a partition are closer to each other than those across partitions, according to the distance metric. We have used Euclidean distance for partitioning all the datasets discussed in this thesis. Our implementations are distributed and multi-threaded and can be used for data from scientific computing and database applications. We have evaluated the partitioner both as stand-alone software and as a module in a full simulation. The raw performance of the partitioner is evaluated using random distributions of points in d dimensional space.

In iterative methods, data is organised on grid (finite-difference, fixed resolution) or mesh (finite volume, with areas of varying resolution) vertices. Many applications in computational science use iterative algorithms which perform some non-trivial computation on the mesh elements and communicate between adjacent elements through a common face. The meshes are usually large enough to be partitioned (decomposed) across multiple processes. In this context, a good partition is defined as one with load balance - with roughly equal work performed at each process and locality - with neighboring cells being co-located on the same processes.

We have used Space-filling curves (SFC) to order data across nodes and within a node. Construction of space-filling curves with good locality is a sub-section of this thesis, although the programmer can define his own rules for producing other permutations. The default space-filling curve in our partitioner is Morton Order, for all dimensions > 1. Specialized Hilbert-like orders are supported for dimensions $2, 3$. The assumption is that *spatial locality* is good, defining distances in higher dimensional space. This has worked well so far for meshes and grids in scientific computing, because the communication patterns are usually nearest-neighbor exchanges. SFCs are a quick and easy method for dimension-reduction in many areas of computer science - databases, networking and parallel computing to name a few. They generate a linear order of points in higher dimensional space, so that successive points on the curve tend to be adjacent in space. The linear list can be used to partition data by slicing it into roughly equal segments and assigning a piece to each process. For mesh partitioning, each mesh element is represented by a point in space, typically 2D or 3D. The center of gravity of the mesh element is its representative point and the mesh is partitioned by partitioning these points. This results in good quality partitions for most meshes. The most commonly used partitioners in scientific computing come from graph partitioning (using the dual graph), i.e Metis [4] and Scotch [5]. SFC orders have also been used for speeding up k-nearest neighbor and point-location problems in database applications which benefit from locality. There is significant difference between database applications and iterative methods, besides the obvious that iterative methods are focussed on solving a particular partial differential equation on mesh elements and database applications perform search, insertions, deletions and updates to an existing database. Database applications and general graphs tend to have higher dimensions (attributes), i.e number of search variables are quite high. The number of permutations increases exponentially with the number of dimensions. A good permutation for a given database and set of search records is harder to find for higher

dimensions. The size of the database and the number of variables, affects sorting, insertion and deletion times which are essential operations. Hierarchical datastructures are used for indexing in multi-dimensional databases [6]. Although the performance degrades with increasing dimensions, they are used extensively at lower dimensions. Databases based on geographic co-ordinate systems use similar methods for searching and indexing [7]. There are other databases using derived distance based metrics for organising data e.g sequences in bioinformatics [8], [9],image processing and multimedia [10],tracking moving objects [11]. Some of these databases use space-filling curves for indexing data, especially Morton order [7], [12].

Several space-filling curves have been described in literature - Hilbert, Peano and Sierpinski are some of them [13]. These curves are defined on regular shapes (squares in 2D and cubes in 3D) with dimensions that are usually powers of 2 (Hilbert, Morton and Siepenski) or 3 (Peano) which makes it difficult to adapt them to irregular point distributions. This can also happen to domains which are not square or cube although points may be equi-distant from each other, for example, circle, sphere and other shapes used in image processing. In short, this work describes a general framework for ordering data using hilbert-like curves and mapping them onto processes and threads on a many-core machine. Our Hilbert implementations are defined for dimensions ≤ 3. SFC order is used hierarchically, first to assign a coarse partition to nodes and later to map finer partitions to cores (threads). Good locality is achieved at all levels. Recursive mapping also reduces data movement and enables quick load balancing for dynamic data (adaptive meshes). Besides generating permutations of mesh elements that have good locality, this work also describes a general framework for organizing large number of random points that are distributed across multiple processes and nodes. The partitioner linearly orders points with good spatial locality across multiple dimensions. The final permutation (order) is distributed across processes. The assignment of processes and threads to permuted data is computed in parallel. It is left to the programmer to re-arrange his data according to permutations generated by the partitioner. Since frequent repartitioning and data transfers maybe expensive for dynamic data, we provide options for incremental adjustments to existing partitions that can be used by applications to find the new location of data. The incrementally refined/coarsened partitions tend to have high similarity to previous partitions, which reduces data transfers. It is possible that a full re-partition might produce a permutation that is entirely different from the previous order. In such cases, the application execution time is likely to increase due to full data exchange between processes. There

is a trade-off here that can be explored. If the overhead of tolerating a poor partition exceeds the cost of full re-partition, it might be beneficial to re-order data from scratch. We have used this approach for meshes in scientific computing, which have edges between mesh elements. It is not clear whether it benefits other testcases.

These are some of the focal points of this thesis :

1. Using geometry to advantage
2. Multi-threaded framework for constructing and managing hierarchical decomposition of space : static and dynamic
3. Generalized space-filling curves in 2D and 3D, that can re-order arbitrary point distributions
4. Parallel Construction of Space-filling curves
5. Low overhead parallel partitions
6. Optimizations in distributed and multi-threaded AMR : datastructures and algorithms
7. Low overhead load balancing schemes for AMR
8. Code restructuring for many-core architectures

Chapter 1 provides an introduction to the problem addressed by this thesis - partitioning of meshes and data. Chapters 2 discusses hierarchical decomposition of datasets. Chapters 3 and 4 describe data re-ordering using space-filling curves in 2D and 3D. The algorithms used, their implementations, and performance evaluation are presented in these chapters. Chapter 5 discusses benchmarks in detail. The partitioner is integrated with benchmark programs and testcases are provided that compare the performance of these programs.

1.2 MESHES

Meshes in scientific computing can be broadly divided into structured and unstructured, in 2D and 3D. This classification is based on the co-ordinate system and the shapes of mesh elements.

1. Structured meshes consist of elements which are usually symmetric, i.e. square or cubes and have the same resolution in all dimensions. Such meshes are used in many applications in scientific computing for which a uniform sample of points is sufficient. They can be partitioned by dividing them into roughly equal blocks because the communication patterns are simple nearest neighbor exchanges. Structured meshes can be stored in contiguous arrays and indexed without pointers. For example, the nearest neighbors of mesh element (i, j) can be computed locally using its indices, i.e. elements $(i-1, j)$, $(i+1, j)$, $(i, j+1)$, $(i, j-1)$ are the immediate neighbors in a 5-point stencil of element (i, j). The same holds for $3D$ meshes, with (i, j, k) indices. Mesh points may be aggregated into blocks to improve cache reuse and pre-fetching. Loops implementing solvers used for these meshes are written entirely using array indices $1...n$ for n points. These programs don't need much re-structuring for many-core and SIMD architectures. An example program for a single iteration of a Poisson process with tiling (blocking) is provided in 1.1. For processors with deep memory hierarchies (at least L1 and L2 cache), block sizes can be chosen to fit a tile each of a_new and a in L2 cache (for example). One could further improve the performance by a second level of tiling for L1 and vectorization [14]. This program benefits from cache re-use because of temporal locality, in addition to spatial locality [15].

Listing 1.1: Structured mesh accesses

```
for ii=1,n,tile_i
 for jj=1,n,tile_j
    for i=ii,ii+tile_i
        for j=jj,jj+tile_j
                a_new[i*n+j] += 0.25*(a[i*n+j]+a[(i+1)*n+j]+
                                      a[(i-1)*n+j]+a[i*n+j-1]+
                                      a[i*n+j+1])
```

2. The loops used for accessing and computing parameters on structured meshes with missing points (holes) are different from dense grids. The mesh elements themselves are used after applying a mask to them, where the mask value at position (i, j) is 0 for missing data and 1 otherwise. Nearest neighbor lookup maybe affected in this case with the addition of an if condition or a bitmask to check the presence of a neighbor in a given dimension. These additional checks can affect some optimizations and increase memory consumption for these meshes. An ex-

ample loop for one iteration of a Poisson process on a mesh with missing points is listed in 1.2. In this loop we have used a mask array to multiply individual elements. Block sizes should now be chosen to fit a tile of a_new, a and $mask$ in L2. Examples for such meshes are structured grids covering water bodies, with missing data on land masses.

Listing 1.2: Structured Mesh with Holes

```
for ii=1,n,tile_i
 for jj=1,n,tile_j
    for i=ii,ii+tile_i
        for j=jj,jj+tile_j
        {
                int num_nbrs=0;
                num_nbrs += mask[(i-1)*n+j]+mask[(i+1)*n+j]
                            +mask[i*n+j-1]+mask[i*n+j+1]
                num_nbrs = max(num_nbrs,1)
                a_new[i*n+j] += a[i*n+j]*mask[i*n+j]
                a_new[i*n+j] += a[(i-1)*n+j]*mask[(i-1)*n+j]
                a_new[i*n+j] += a[(i+1)*n+j]*mask[(i+1)*n+j]
                a_new[i*n+j] += a[i*n+j-1]*mask[i*n+j-1]
                a_new[i*n+j] += a[i*n+j+1]*mask[i*n+j+1]
                a_new[i*n+j] *= mask[i*n+j]*(1/num_nbrs);
        }
```

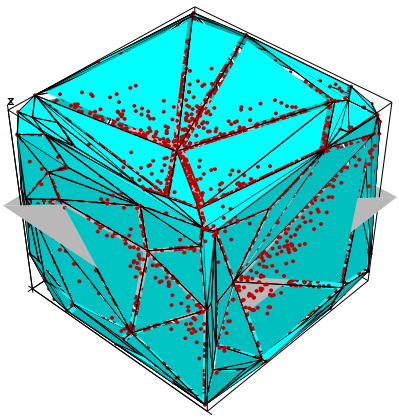


Figure 1.1: Unstructured 3D Mesh with tetrahedral elements

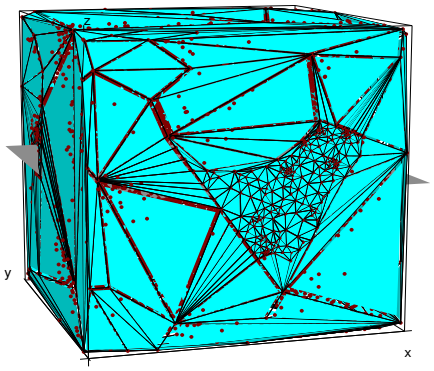


Figure 1.2: Unstructured 3D Mesh with tetrahedral elements - Refined

We add a category of meshes here which are used in block-structured AMR ap-

plications and cosmology simulations [16]. These meshes may be constructed from finite difference grids which require higher resolution in certain areas of the domain. Computation and communication are performed at the granularity of blocks instead of mesh points in these problems. A *block* is an axis-aligned hypercube of mesh points that are smaller than the cache. Each block is treated as an element for partitioning. Computations are performed on all points contained in a block, which have spatial and temporal locality due to its size. For block-structured AMR, the areas of the mesh marked for refinement are sub-divided equally in all dimensions, unlike fully unstructured meshes where the refinement ratio may differ in each dimension. There are additional constraints on the refinement levels of adjacent mesh elements to allow smooth transitions - No two adjacent mesh elements should differ by more than one refinement level. Since mesh points within a block are equi-distant, memory accesses in loops are similar to structured meshes, which gives good performance on many-core architectures. One may need indirect addressing or lookup-tables to locate non-local blocks in these meshes. Cosmology simulations [17] are modeled using a random distribution of points that tend to be clustered in certain areas. However, the meshes used are similar to block-structured AMR with uniform refinement in areas of clustering. The particles within a mesh block are usually stored in contiguous arrays and particle-particle (particle-cell) interaction loops can be structured to enable vectorization.

Listing 1.3: Unstructured Mesh Accesses

```
for i=1,n
{
  a_new[i] += a[i]
  for j=1,num_nbrs(i)
  {
     a_new[i] += a[nbrs[i][j]]
  }
}
```

Unstructured meshes consisting of triangle or hexagonal elements in 2D and tetrahedral elements in 3D are treated as special cases. The neighbor lists for these mesh elements have to be stored explicitly. Therefore, nearest neighbor exchanges require additional levels of indirection. The meshes may be stored in

contiguous memory, but neighbor lookup can result in more irregular memory accesses than in structured meshes. Indirect addressing increases cache misses in the solver loops. Unstructured meshes may be uniformly refined, where all elements have the same size and number of neighbors or highly irregular with several regions of coarse and fine elements scattered in the domain. Two examples for unstructured meshes in 3D are shown in figures 1.1 and 1.2. A simple loop for a single iteration of a Poisson process on an unstructured mesh is provided in listing 1.3. Neighbor indices are stored in adjacency lists. There is one level of indirection for each neighbor in the inner loop. One could improve the performance of this loop by unrolling the outer loop. Meshes that use indirect addressing in this manner are classified as unstructured in this work.

1.3 THE PARTITION PROBLEM

Mesh partitioning is a subset of the larger problem of graph partitioning. The balanced graph partition problem known to be NP-complete, can be formally defined as: given a graph G with vertex-set V and edge-set E, both weighted, a P-way partition of the graph should create P disjoint subsets minimizing the maximum weight of a partition or total edge-cut (communication volume) or maximum edge-cut. This is often formulated as an optimization problem with an objective function and a set of constraints. A commonly used objective function is minimization of maximum edge-cut or communication volume, and constraints are placed on the maximum load imbalance between partitions. There are more complex formulations with objective functions minimizing the maximum number of outgoing or incoming messages, weighted sum of multiple criteria, etc. A couple of different formulations for this problem can be found in [18]. They also provide options for the programmer to form his own objectives and constraints.

We discuss below, the objective function that minimizes maximum communication volume, subject to load imbalance constraints. Let e_i be the sum of weights of outgoing edges of any partition p_i, which contribute to the total communication volume. The objective function can be formulated as:

$$\min_{i=1}^{P} \max e_i \tag{1.1}$$

For a given partition set, let w_i be the load (sum of the weights of elements) of any partition p_i. Define load imbalance as the maximum difference between weights of any two partitions p_i and p_j. The R.H.S in the constraint is the maximum desired value for load imbalance, say X.

$$\max_{i=1,j=1}^{P} (w_i - w_j) \leq X \tag{1.2}$$

Solutions to the partition problem are broadly classified into Geometric, Greedy, Combinatorial optimization, Spectral and Multi-level methods. Except for geometric partitioners, most of these methods are iterative, based on heuristics and computationally expensive.

Approaches to solve the partition problem can be broadly classified into

1. Exact and approximate algorithms

   There are some algorithms that solve the exact bi-partition problem or relaxed versions of it [19], [20]. But running times for these algorithms are quite high. Bi-partition of grids without holes can be solved optimally in $O(n^4)$ if there are n vertices. Approximation algorithms for bisection of unweighted planar graphs that are $O(logn)$ optimal are also described in literature [21]. However these algorithms are not used in practice. Min-flow max-cut implementations belong to this category [21]. The objective functions in these algorithms is to minimize max-flow between two partitions. These implementations tend to find natural cuts in graphs, which are likely to be imbalanced. Load balance is added as additional constraint or incorporated into the cut-metric [21]. PUNCH [22] is a multi-level algorithm that computes min-cuts in road networks with added load balance constraints. The advantage of PUNCH over other partitioners is that it is a parallel implementation that has shown good performance on large graphs of the order of tens of millions of vertices.

2. Order producing

   Graph coloring is a commonly used technique to partition and order columns of sparse matrices [23]. The sparse matrices may be derived from graphs, which makes them a good category of partitioners for general graphs. There is a difference in the objective functions for graph coloring vs graph partitioners. The objective here is to minimize the number of colors used. Typically, there are no

constraints on the number of vertices having the same color or the coloring of adjacent vertices. This is a very fast technique to order vertices of a graph, commonly used for task assignment in schedulers where graph vertices are the tasks in the algorithm. Coloring is not a good choice for partitioning meshes because load balance and edge-cut are not usually minimized in this method. *Colpack* is a recent implementation of graph coloring algorithms that has shown good performance [24].

3. Spectral methods

   Spectral methods partition graphs by computing the eigen vectors of a Laplacian derived from the adjacency matrix of the graph. For a graph G with vertex set V and edge-set E, both weighted, the Laplacian L is defined as :

$$L = D - A \tag{1.3}$$

   where $A \in R^{nXn}$, is the weighted adjacency matrix of the graph and D is the diagonal matrix.

$$D = diag(Ae) \tag{1.4}$$

   $e = (1, 1, ..., 1)^T$

   For p partitions, the first p eigen vectors of the Laplacian are chosen as the partition order of the matrix (graph). Minimization objectives and constraints can be incorporated into to this method [25], [26] to improve the quality of partitions.

   Spectral partitioners produce good quality bipartitions although they are computationally expensive. They are typically used as routines in other graph partitioners for bisection or for partitioning smaller hypergraphs. There are some recent parallel implementations of spectral partitioners that are faster, but results are shown for small numbers of partitions [27].

4. Global and local search based

   There are several iterative graph partitioning algorithms which refine partitions using heuristic functions. One of the simplest ones is Kernighan-Lin [28] which searches for a global solution using local exchanges. There are many variants of this algorithm. A linear time implementation of Kernighan-Lin's algorithm [29] is

widely used by many partitioners. Such algorithms are often used as refinement routines to improve the quality of existing partitions. There are some trade-offs though. For example, it is difficult for local searches to satisfy load balance constraints along with minimum edge-cuts in the mesh. A pair-wise exchange that lowers load imbalance may increase communication cost and vice-versa. This depends on the quality of the initial partition. If the initial partition is good, then refinement will quickly converge to a better solution. Some implementations of these algorithms make global moves to achieve a global minima. There are others that restrict movement between neighbors, in which case you may not find a global solution. The implementations of these algorithms add conditions to avoid getting trapped in local minima. More details are provided in the section on multi-level methods.

5. Bottom-up

   There are implementations that use bottom-up approaches called seeding or graph-growing to compute initial partitions [30]. Random seeds are picked in the domain. Search routines like breadth-first search are spawned at these locations to determine membership of a vertex in a partition. Seeding terminates when the partitions satisfy load balance constraints or when all vertices are assigned to partitions, which ever happens first. There is no guarantee on the quality of partitions. One approach is to create several partition sets and pick the best from them. The other option is to refine partitions using one of the iterative schemes mentioned earlier, i.e. [29].

6. Multi-level

   Discussed in detail in section 1.4.

7. Geometric

   Geometric partitioning algorithms use spatial co-ordinates of the vertices of a graph, ignoring connectivity information. [31] has some techniques for partitioning points in 2D and 3D. These partitions tend to have poor edge-cuts compared to the other methods described here. Since space-filling curves fall into this category, we discuss these methods in detail in the later sections. We present empirical results for both structured and unstructured meshes for comparison with graph partitioners. This method is particularly good for parallel implementations due to low overheads - both memory and execution time. It has shown

promise as an efficient adaptive scheme that lends itself well to rapidly changing workloads. This method can be extended to general graphs with very large vertex degrees and edges [7].

8. Streaming algorithms

   One of the recent developments is the addition of streaming versions of partitioning algorithms intended for handling large datasets. [32] describes an implementation that partitions data based on some metrics defined on the adjacency graph. The program is sequential and maintains p bins corresponding to p partitions. The overheads are less, data is accessed once and assigned to bins without accessing other graph vertices/edges. The assignment of vertices to bins ranges from random to carefully chosen metrics that pick the vertex with most number of shared edges. The assignment overheads depend on the metrics. For example, to assign a new vertex v to the bin with most shared edges, it is necessary to access all previously assigned vertices in a bin and count the number of edges shared with v.

Most of these implementations are sequential. Typically, meshes in scientific computing and databases are large and stored across multiple compute nodes. The applications using these meshes/datapoints require quick parallel computation of partitions. An additional caveat is dynamic workloads that change during a simulation. Simulations using dynamic or adaptive meshes require re-partitioning which brings the partitioner into the simulation routine. The partitioner is no longer an isolated entity, instead, it is online (part of some iteration). An online-partitioner could potentially slow down the application, depending on its overheads. Some of the widely used parallel partitioners belong to the class of multi-level methods. They have shown good scaling with increasing number of processes. Nevertheless, re-partitioning and data migration are non-trivial overheads introduced by parallelism, which need to be handled well. There are several questions here that need to be answered - how often to re-partition, where to include the partitioning routine in a simulation etc. We attempt to answer some of these questions through benchmarks in chapter 5.

In recent times, there has been a lot of work on developing packages for processing large real-world graphs, typically derived from social networks, web graphs etc. These are random graphs that follow the power law degree distribution [33]. The graphs are large, with hundreds of millions of vertices on an average and billions of edges. The average degrees of these graphs are quite high which makes them different from

meshes. Most of these graph processing frameworks are distributed and provide interfaces for handling large files, searches and data transfer. Depending on the application, the graphs may be fixed or dynamic. Dynamic workloads allow addition and deletion of vertices and edges in the graph. The problem of graph partition still holds for this domain - generate good quality partitions where neighbors are co-located on the same process. Most of the packages use hashing to map vertices to partitions. A set of bins are assigned to processes and threads. Hash functions are usually random, based on some permutation of bits. The performance of these partitions was found to be very poor and this led to the use of graph partitioners like Metis and Parmetis [4] in this domain. Partitions with better locality improved the performance of these applications considerably. Giraph++ [34], GraphX [35], Dryad [36], Naiad [37], DistGraphLab [38], Mizan [39] and Pregel [40] are some of the widely use packages for graph analytics on real-world data. Naiad uses space-filling curves to partition adjacency matrices (edges) of graphs. This can work depending on how graph vertices are numbered. Different numbering can give rise to different space-filling curves, some of which are better than others. We support partitioning of data if it can be embedded in d-dimensional space. We leave the embedding to the programmer because distances and relationships between graph vertices are application dependent. For example, if there are many parameters per vertex, one can resort to a detailed statistical analysis and pick the d most significant ones and treat them as d dimensions [41]. There are many advantages to using a geometric partitioner for general dynamic graphs - they are usually fast and have low overheads.

Meshes in iterative methods also appear as sparse matrices derived from their adjacency graphs. There are many methods in linear algebra to re-order these matrices to enable computation, both sequential and parallel. For a given permutation of mesh elements, its adjacency matrix is a sparse matrix with non-zeros used to indicate the presence of an edge between any two elements. The matrix has zeros elsewhere. If edges are unweighted, all non-zeros will be equal to one. Otherwise, they can take any positive real value. There are some methods which improve the sparse nature of these matrices for speeding-up computation. The objective is to convert the sparse matrix with a random distribution of non-zeros to a banded or diagonal matrix which can be partitioned into blocks. Some of these methods introduce additional zeros while re-ordering and partitioning matrices. These are called fill-in values. The problem of finding the best order with minimum fill-in is NP-complete. Commonly used matrix reordering algorithms are e.g nested-dissection [42], [43] and reverse cut-hill [44].

There are variations of these algorithms that are parallel. A recent algorithm uses a geometric method to create coarse partitions, followed by nested-dissection at the lower levels to re-order matrix elements [45]. Space-filling curves can be used to create coarse partitions from sparse-matrices. However, we do not re-order the rows and columns of matrices. The non-zeros in a sparse computation can be distributed across processes and threads using space-filling curves for load balancing and spatial locality.

1.4 MULTI-LEVEL METHODS

Multi-level graph partitioners like Metis [18] and Scotch [5] are widely used for partitioning general graphs and meshes. They do considerable work to optimize and improve the quality of partitions. We have used these methods as baseline for all our measurements. As the partition problem is NP-hard, it is difficult to find a good baseline implementation against which other algorithms can be compared. Since our methods are single pass without iterative improvement, we consider multi-level methods a good choice for baseline comparison of partition quality. Between Metis and Scotch, these packages have similar frameworks, although the algorithms, implementation and tuning parameters tend to differ. Most of them are optimized for planar graphs, i.e meshes while others are more general and capable of handling large real-world graphs [46]. Some of the multi-level packages have default objective functions, while others like Metis and Scotch offer tuning parameters to the programmer which can be adjusted to pick an objective function and set of constraints that best describes his requirements. Most applications in scientific computing benefit from minimizing the maximum communication volume and load imbalance. Real world graphs could gain from minimizing the maximum degree subject to load balance constraints. There are options to add weights to nodes and edges. It is left to the discretion of the programmer to carefully understand the requirements of his application and tune the partitioner.

Multi-level graph partitioning is outlined in this section. Let G be any input graph with vertex set V and edge-set E. Let n be the number of vertices, each of weight w_i, and m the number of edges with weight e_{ij} assigned to edge (i, j). Let p be the desired number of partitions. The different stages in a k-way multi-level partitioner are explained below:

1. Graph coarsening: The input graph is coarsened over many stages into a smaller hypergraph of a few hundred hyper nodes. Each coarsening stage uses edge matching, either randomized or sorted (according to edge weights) to merge vertices [47], [18]. Coarsening is also implemented by seeding multiple Breadth-first Search (BFS) in the mesh and greedily adding vertices to these partitions. The widely used implementations for coarsening are based on matching. The coarsened hypergraph is partitioned into k-sets using a more expensive algorithm, like a spectral method [25]. Some packages prefer recursive bisection to direct k-way if k is a power of 2.

2. Graph refinement:

   Coarse graph partitions are iteratively improved by projecting them onto finer partitions, until there are p partitions. The objective function is satisfied during every iteration of refinement, without violating constraints. There are several algorithms for continuous improvement of partitions in Metis, some of which include Fidducia-Mattheyas [29], a variant of Fidducia-Matheyes and greedy [18]. Refinement algorithms improve the quality of partitions by searching for solutions that satisfy the objective function and constraints. The goal is to obtain a better refined partition from a good initial partition. The number of refinement stages is a tuning parameter. The algorithm terminates naturally when no further improvements are possible or when a fixed number of iterations are reached. Refinement algorithms differ in their choice of data structures and minimization criteria. Two additional refinement algorithms implemented in Scotch are Band [48] and diffusion [49].

3. Mapping to a process topology: This is an optional step that is found in some partitioners. Partitions are eventually mapped onto a machine or MPI ranks. The machine architecture or virtual topology of MPI processes is represented as a graph. These graphs are partitioned into clusters using similar techniques or simpler methods like recursive bi-partitioning. Vertices are assigned numbers according to clusters, and mesh partitions are assigned to these vertices, in the order of indices. Topology can be extended further to include memory hierarchy such as NUMA regions within a rank. Different weights are assigned to each type of vertex in the topology graph. Scotch has an extension that handles topology-aware mapping of partitions to nodes and cores. If using Metis, mapping is left to the programmer. A connected graph specifying the machine topology with edge-

weights that model communication overheads can be given as input to Metis and its output used to map partitions into clusters of nodes and cores.

Since most coarsening and refinement algorithms are randomized, it is advised to generate multiple partition sets and choose the best partition. Each execution of the coarsening stage will initialize the refinement stage to a different configuration, some of which may be worse than others. We have used these partitioners with default parameters for most of our experiments.

Jostle [50] is a parallel graph partitioning software that implements a multi-level partitioning algorithm, like Metis and Scotch.

The Zoltan package [51] provides interfaces to both hypergraph partitioners Metis and Scotch along with a custom hypergraph partitioner [52]. Other multi-level partitioners include Kahip [53], Party [54],Patoh [55] and DiBAP [56]. Patoh has used agglomerative clustering during the coarsening stage to form hypergraphs. They have also used several variations of Kerninghan-Lin and Fidducia-Matheyses for refinement. The quality and partitioning times are comparable to other multi-level implementations. Kahip [53] implements a faster coarsening algorithm, called Global Path Algorithm, compared to Metis. The refinement algorithms have been retained along with a couple of new implementations for local exchanges. One of the refinement schemes is based on seeding two BFS until the bi-partitions satisfy load balance criteria. The BFS is done repeatedly along with vertex exchanges to lower edge-cut. The second implementation is a k-way version of Fidducia-Matheyes. There is some minor improvement in performance. There is also a parallel version of this implementation, with performance comparable to Parmetis [57]. Party [54] is an attempt to improve the refinement phase in multi-level methods. The coarse hypergraph is partitioned using recursive bisection instead of direct k-way method. The partitioned hypergraph is improved iteratively by a new refinement algorithm called *helpful-sets*. This algorithm is based on Fidducia-Matheyses with variations in the potential function and data structures used. This could lead to differences in the number of iterations and exchanges to find an improved solution. The improvements have not been evaluated fully. They have a multi-threaded version, which is compared with Jostle and Parmetis. They get better edge-cuts for structured symmetric meshes with finite degrees, which was the objective of the partitioner.

DiBap [56] is another multi-level partitioner with additional algorithms for coarsening and refinement. The coarsening phase includes a faster geometric method that

computes the distance between the center gravity of a partition and a new vertex for membership. The vertex gets assigned to the partition that is closest to it. The last few iterations of coarsening use this method, also known as, Bubble-FOS. New additions to the refinement stage are bubble-based diffusion and truncated-diffusion (TRUNC-CONS). Both are based on diffusion, which transfers load from heavily loaded processes to lightly loaded ones. Only those moves are encouraged that maintain load balance. Refinement stages also consider minimizing edge-cut while choosing the pair of processes engaged in diffusion.

A detailed discussion of Metis and Scotch can be found in Chapters 3 and 4. We give the pros and cons of using multi-level partitioners along with testcases.

CHAPTER 2: PARTITIONING DATA USING GEOMETRY

2.1 OUTLINE

In this chapter, we discuss the software architecture of the partitioning framework. There are many benefits to using geometry with data. It provides a natural way to organize data relative to each other. This is exploited greatly in meshes, but the same methods can be extended to other point distributions. One method is to use a mesh generation software to impose a mesh on a pointset - Voronoi tessellations [58], Delaunay triangulations [59]. Once the mesh is generated, its elements can be permuted and partitioned using a suitable graph partitioner. One can also use geometric methods to partition pointsets without edges. Our partitioner does not distinguish between pointsets and meshes.

The input to the partitioner is a set of points and a distance criteria in d dimensional space that can be used to separate them. We provide the outline below :

1. Construct a hierarchical decomposition of space enclosing the dataset using axis-parallel hypercubes (boxes).
2. Define a traversal on the boxes that preserves locality. The order of traversal generates the final permutation of data.
3. If there are P partitions, the permutation list is sliced into P roughly equal length segments.
4. If the dataset has unequal computation at different regions in the domain, weights are assigned to all points. In the current implementation, this does not affect the decomposition or the traversal. In the weighted case, partitions are still roughly equal length segments, where the length of a segment is the sum of weights of its constituent points.

We have used kd-trees for hierarchical decomposition of the domain. In our current implementation, there are two possible traversals - Morton and Hilbert. Our implementation of Morton is a generalized version that is independent of the shape of the domain. It is defined on all dimensions, $d >= 2$. The Hilbert curve is also a general version which can be applied to asymmetric domains. In the current version, Hilbert

curves are defined only on 2 and 3 dimensional data.

In the remaining sections in this chapter, we discuss and evaluate these components in isolation. The first section deals with kd-tree implementations. The general description is provided along with various optimizations for a parallel environment. Here we have discussed shared memory programming in some detail. The algorithms for splitting and the data structures used are discussed. Various test cases are provided to compare the performance of different parallel implementations. The test cases are random point distributions in this section - not derived from meshes. But the implementation is agnostic to the source of data. They may or may not be derived from a mesh. After a discussion on kd-trees, we present rules for Hilbert traversals - 2D and 3D. The traversals are evaluated empirically by comparing against multi-level graph partitions.

2.2 METRICS FOR EVALUATING PARTITIONS

The partition problem was discussed briefly in Chapter1. In this section we define a minimal set of metrics that we have used to evaluate partition quality. These metrics appear later while evaluating the performance of testcases. The metrics define the way in which measurements are taken at various points in programs to evaluate performance.

Define work, the amount of computation per mesh point and communication. Computation per mesh point is also referred to as *weight* in some descriptions. We isolate these operations in a program and define metrics separately. This separation is essential for reasonable evaluation of parallel programs. Both computation and communication depend on the number and shape of partitions.

The effort taken to optimize programs depends on the ratio between computation and communication. For programs with compute heavy code sections and sparse communication, it may be worth the effort to restructure the code to reduce cache misses, re-use data and even think of simpler equations that can perform the same computation. In all cases, the criteria that works across optimizations is load balance.

The communication graph of a parallel program consists of vertices that are code sections executed by processes or threads and edges that are weighted according to

the number of bytes exchanged between these code sections. There are three metrics that are easily defined on the communication graph - maximum degree, sum of edge weights and maximum edge weight. At any given point in time, a subset of code sections (vertices) that satisfy dependencies are executed in parallel. Therefore, between sum and maximum, the more significant metric is the maximum edge weight. A program can have different communication graphs at various points. The communication graph is updated when the dataset is updated or if the execution of a vertex generates new vertices, e.g. adaptive computations and graph traversal.

Minimum computation time is obtained when the load distribution is roughly equal across partitions. If the distribution is not balanced, there is some process p_i that has more load than others. Let T_{comp} be the total computation time of a parallel program with P partitions. Processes are numbered from $0, ..., P-1$ and let w_i be the load of any partition p_i Let C be the amount of work per unit weight.

$$T_{comp} = \max_{i=0}^{P-1}(C * w_i) \tag{2.1}$$

When there is imbalance in the load distribution, heavily loaded partitions take more time to complete than others, which increases the maximum computation time of the application. On the other hand, lightly loaded processes spend time idling at barriers. An ideal distribution has the same weight on all processes and this should be the mean value.

We have used a similar model for inter-process communication where each process has degree d_i and communication volume e_i in bytes. Assume a PRAM delay model for communication where α is the setup cost or latency of a message and β is the cost of transferring a byte on the network. If a node has a single-port, the total communication cost of a partition is the sum of all individual messages that are sent/received. The communication cost of the entire group is the maximum communication cost of any partition. Let T_{comm} be the communication cost of a communication phase where point-to-point messages are sent between processes.

$$T_{comm} = \max_{i=0}^{P-1}(\alpha * d_i + \beta * e_i) \tag{2.2}$$

Parallel programs exhibit different behavior depending on the frequency and proportion of computation and communication. We have used the following models for

measuring execution time in our programs. Suppose there are a computation phases and b communication phases in a program. Let T_{BSP} be the total execution time of the program if it follows a BSP [60] model where computation and communication are carried out in stages, and processes synchronize between them. Then,

$$T_{BSP} = a * T_{comp} + b * T_{comm} \tag{2.3}$$

Suppose there are r regions in a program where computation and communication overlap - there is no synchronization between processes until the program is terminated. Let T_{comp_i} be the computation time and T_{comm_i} the communication time of process i. The total execution time when these operations overlaps is $T_{overlap}$, which is defined as

$$T_{overlap} = \max_{i=0}^{P-1} r * max(T_{comp_i}, T_{comm_i}) \tag{2.4}$$

The assumption here is that $T_{overlap}$ is equal for all r stages. Most often parallel programs are a mix of both styles. There are computation and communication phases and code sections that are asynchronous. Let T_{total} be the total execution time of a parallel program,

$$T_{total} = a * T_{comp} + b * T_{comm} + \max_{i=0}^{P-1} r * max(T_{comp_i}, T_{comm_i}) \tag{2.5}$$

2.2.1 Multi-threaded and Hybrid Applications

Similar metrics can be used for multi-threaded and hybrid applications where a group of threads are spawned for computation within a process. We have followed these metrics while measuring execution time in our multi-threaded and hybrid programs. Let P be the number of processes numbered from $0, ..., P-1$ and T be the threads per rank, numbered from $0, ..., T-1$. Let w_i be the computational load per rank i that is distributed among T threads. If the total computation per node is distributed equally, $t_i = \lceil \frac{w_i}{T} \rceil$ is the load per thread. We can use the same argument above in favor of a balanced load distribution between threads. No thread in the group should be overloaded and no thread should idle at barriers. Maximum degree and

edge-cuts metrics are analogous to the number of synchronization points and cache misses in a shared memory program. Access to shared data-structures is controlled using mutexes. Similarly, the execution of threads can be controlled using signal-wait constructs. The number of such primitives in the program depends on the partitioning and ordering of data. If there are K synchronization points in a program and at most T contenders per access (worst case), we can form a metric for communication cost that depends on program structure. Let k be the average cost incurred by a thread for gaining exclusive access to a location. Another influence here is cache misses. Let B_i be the average cache size available to thread T_i. If the thread accesses M_i distinct bytes in total, there are at most $c * \lceil \frac{M_i}{B_i} \rceil$ cache misses. The overhead factor $c, >= 1$ depends on the cache mapping policy and number of hardware contexts on the same CPU. Let α be the computation cost per unit weight and β be the cost of transferring a byte from memory to cache. The computation cost per rank T_{comp_i} is now split into computation and synchronization costs incurred by a group of threads.

$$T_{comp_i} = \max_{i=0}^{T-1}(\alpha * t_i + \beta * c * \lceil \frac{M_i}{B_i} \rceil) + \sum_{i=0}^{K} k * T \tag{2.6}$$

One can define more complicated models where the cost of data transfer depends on the type of memory used (NUMA effects). We have kept it simple for now.

Orders that reduce communication costs, i.e. cache misses and synchronizations are better *partitions* in shared memory. We have used space-filling curves for ordering data within ranks. For a given data layout, one can always find orders that are competitive, with fewer cache misses. Similarly, there is at least one permutation that has the highest number of misses. We have not used counters to measure cache misses, but the differences in execution time are used to justify choice of data structures and favor one data layout over the other. Concrete examples are discussed in the section on parallel AMR 5.3.

2.3 KD-TREES

Unstructured data is difficult to represent, index and manipulate, both in scientific computing and database applications. There are many reasons behind this, the ma-

jor factors being the size of the dataset and the number of dimensions and features attached to a point. This difficulty increases with increase in the number of dimensions, which has not been fully addressed so far in parallel computing [61], [62], [63]. There is plenty of work that deals with handling large datasets efficiently, especially in databases [64]. One of the commonly used data structures for representing multi-dimensional datasets is a kd-tree [65]. It is mostly used for point location, nearest-neighbor searches and region searches. They are not so popular in scientific computing for representing meshes. Moab [66] has a mesh implementation that uses kd-trees for storing and coupling different meshes. There are variants of this data structure in cosmology and AMR applications which handle dynamic datasets e.g. Barnes-hut [17]. Parallel KD-trees are widely used data structures in 2D and 3D animation, for locating objects and computing interactions. For such applications that are real-time, it is essential the data structure is concurrent and has low synchronization overheads. These programs should use a processor's memory hierarchy efficiently by structuring code for spatial and temporal localities. Ray-tracing [67] and collision-detection [68] software both of which are real-time simulations belong to this category. They typically have dynamic scenes which are continuously updated.

These applications and their implementations are typical testcases for GPU performance, therefore, there has been considerable effort in the past to optimize them on SIMD [69]. Several optimizations for kd-trees are discussed in [70] which improved their SIMD performance. The most commonly used splitters in these trees are based on volume-surface area criteria, also known as SAH-trees [69]. These are useful metrics for meshes with well-defined adjacencies and communication patterns. Testcases for general graphs and random distributions of points without adjacencies haven't been addressed in these papers. The dimensions are usually restricted to 2D and 3D and simulations are iterated over time, continuously updating scenes and interactions.

We have relied heavily on the geometry of the outer-most bounding box and less on the shape of individual mesh elements. This was done intentionally, to remove dependence on mesh types and make the partitioner capable of handling random points with clusters. There are some similarities between GPU specific implementations and ours, mostly in the optimizations used. Both implementations are synchronous, based on single-sweep techniques, where threads independently sweep through the data, measuring metrics that are aggregated at the end of the sweep. These metrics are used for decisions later. Our synchronization methods are different from GPU thread-

ing constructs. CPUs do not have the additional restrictions that hardware threads on GPU have - with respect to warps and thread blocks. We first describe the sequential algorithms and later explain their parallel equivalents. Multi-threading constructs, synchronization methods and parallel overheads are explained in detail in section 2.4.1.

2.3.1 KD-tree Construction

Algorithm 2.1 KD-tree Construction Method

```
1:  procedure KDTREE_BUILD(node n)
2:      if then n.size() <= bucket_size
3:          return
4:      else
5:          dim ← n.dim_of_max_spread()
6:          n.split(dim)
7:          n1 ← n.lower()
8:          n2 ← n.upper()
9:          if then n1.is_valid()
10:             KDTREE_BUILD(n1)
11:         end if
12:         if then n2.is_valid()
13:             KDTREE_BUILD(n2)
14:         end if
15:     end if
16: end procedure
```

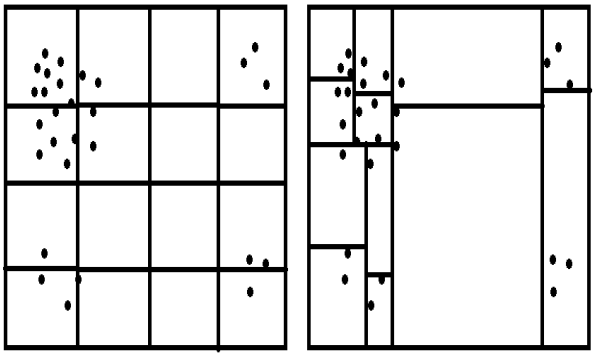


Figure 2.1: Two examples of kd-trees from random point distributions, a: midpoint splitters with cycling through the dimensions b: median splitters along the dimension of maximum spread

The most common implementations of kd-trees have two types of nodes - non-

Algorithm 2.2 Node Split

```
1: procedure SPLIT(dim)
2:     value ← Median
3:     set_splitter(dim, value)
4:     n ← datasize
5:     n1 ← lower()
6:     n2 ← upper()
7:     for doi ← 1, n
8:         if thendata[index[i]].coord[dim] ≤ value
9:             n1.add(index[i])
10:        else
11:            n2.add(index[i])
12:        end if
13:    end for
14: end procedure
```

terminals and terminals (leaf nodes). At the non-terminal nodes, the dataset in R^d is divided into subsets by splitting along a hyperplane in $d-1$ dimensions. We have used two degrees of freedom while deciding a splitting hyperplane - a splitting dimension perpendicular to the hyperplane and a splitting value along that dimension. We have restricted ourselves to hyperplanes that are axis-parallel. The subsets are assigned membership to two sub cells depending on the position of points relative to the splitting value. Suppose the splitting dimension is i and the splitting value is m, then all points with co-ordinate values less than or equal to m along the dimension i are assigned to the $lower$ sub cell and the remaining points belong to the $upper$ sub cell. A node stores information regarding its splitting hyperplane. The choice of hyperplanes affect the maximum depth of the kd-tree, its size (number of nodes) and time taken for tree construction. This has been explained in detail in [65]. The pseudo-codes for our implementation are described in algorithms 2.1 and 2.2. We have used a two-level data structure to reduce repeated irregular accesses to mesh data. The first level consists of a geometric summary of the mesh maintained using an index vector and a vector of co-ordinates in some order.These are the only vectors the partitioner accesses. The co-ordinate vector is never modified, all accesses are reads. The vector that is continuously updated is the index vector. We have allocated extra space for copying in the index vector. If there are N points in d dimensional space, the input to the program is $N*(d+1)$ doubles for storing d co-ordinates and one weight value per point, along with $2*N$ unsigned integers for storing and copying indices. These data structures and their accesses are explained in detail along with the parallel kd-tree implementation in

the next section. We first provide a discussion of the splitters used.

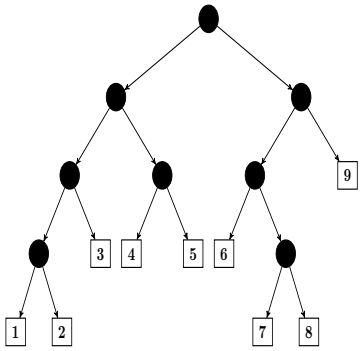


Figure 2.2: KD-tree of a set of points in 3D

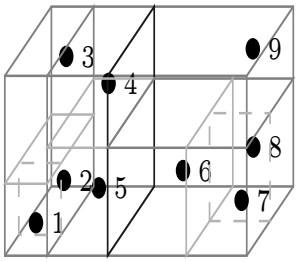


Figure 2.3: Dataset with splitting hyperplanes

The diagrams in figure 2.1 show a random distribution of points in 2D and two ways of separating them with hyperplanes perpendicular to the X and Y axes. The diagram on the left uses a midpoint splitter where splitting dimensions are fixed $\{0, 1\}$, in that order. Notice a large proportion of empty sub cells when using this set of splitters for separating points. The figure on the right has the same point distribution but they are split along the dimension of maximum spread and the splitting value is the median co-ordinate. The diagrams in 2.2 and 2.3 show a random distribution of points in 3D split using 2D hyperplanes and its corresponding kd-tree.

The partitioner takes co-ordinates of the unstructured dataset (including finite difference and finite element meshes) as input and constructs a kd-tree. One of the bottlenecks for large data is tree construction time which includes computing splitters and data movement. We have reduced the execution time for sequential versions by using linear or constant time algorithms for computing splitters and maintaining offsets to the index vector instead of explicitly copying them to non-terminal nodes. The parallel kd-tree implementation is explained in detail in later sections. The output of the partitioner is a final permutation that may be used by the application for re-ordering data accesses to the dataset.

The total cost of building a kd-tree from a random distribution of points depends on the splitters and the number of levels required to separate points in the input distribution. We have included buckets for specifying the granularity of leaf nodes. The size of a bucket in turn influences the maximum depth of the tree. Every leaf node will have at most $BUCKETSIZE$ elements. For an input dataset with N points in d dimensions and $BUCKETSIZE = 1$, the worst case time taken to build the tree is $O(N^{d+1})$ where

the maximum depth of the tree is N. This can happen with highly unbalanced cuts which split the domain of N points into two sub cells, each containing $N-1$ and 1 points. The total number of splits required to separate N points is therefore N. The choice of dimensions is another variable for splitters. For example, there can be two points which are exactly equal in $d-1$ dimensions. In this case, for a fixed order of dimensions, one will have to try $d-1$ cuts, before these two points are separated. This adds the exponent d to the term. The average and best costs are obtained for splits that are balanced. If the splitting dimensions follow a fixed order, then the maximum depth of the tree is $O(logn)$ in each dimension. The total time taken for tree building is at most $O(N*(logn)^d)$. For optimal constructions, we pick the dimension of maximum spread at every node. The maximum depth of the tree will not exceed $O(logn)$ for these trees. For all distributions, median splitters are consistently better than midpoint splitters in obtaining balanced trees of depth at most $O(logn)$. The cost of splitting or computing the median can be made linear if done carefully. Therefore, for d dimensions, the cost of tree building is $O(N*logn)$.

For large datasets, there is a trade-off between the complexity of choosing a splitting hyperplane and the size of the tree. We have used a couple of low-overhead splitters that avoid empty nodes in the kd-tree.

1. Midpoint of the dimension of maximum spread : This splitter first computes a tight bounding box around the points within a node. The longest dimension of the bounding box is picked. The splitting value is the exact midpoint of co-ordinates along this dimension. It takes linear time $O(N)$ to compute the extents of each bounding box and determine the dimension of maximum spread. The extents for a node are updated incrementally by sweeping through its points once.

2. Exact Median of the dimension of maximum spread : The distinct co-ordinates of all points in the domain are stored in separate vectors, one for each dimension. These vectors are pre-sorted once before tree building. For d dimensions, the cost of pre-sorting co-ordinate vectors is $O(d*N*logN)$. A tight bounding box is computed around the points in the domain and the longest dimension is picked. The median value of the co-ordinates in this dimension is the splitter. Co-ordinates of points in each node cover non-overlapping sections of co-ordinate vectors. The co-ordinate ranges in all dimensions are stored in tree nodes. When a node is split along dimension i, two sub-cells are created. Both sub cells inherit ranges from the parent cell for all dimensions except i. For dimension i, the range is

split at the median which is located at $(max - min)/2$ where max and min are the extents of i. All this computation can be done in constant time. The cost of computing the exact median is $O(d * N * logn)$, which is the cost of pre-sorting.

3. Approximate Median of dimension of maximum spread : This implementation is used if the number of points is large enough that pre-sorting co-ordinate vectors in d dimensions becomes an overhead. Instead of sorting the entire co-ordinate vector, a smaller subset of co-ordinates is sampled from every thread for every dimension. These subsets are gathered and sorted. The approximate median is the median of the sample set. The approximate median works for large random distributions of points when sufficient number of samples are picked. We used this implementation for some of the testcases. The sorting overhead still holds. Moreover, in this implementation sorting and re-sampling are done at every node when it is split, compared to the previous implementation where pre-sorting was sufficient. The cost of sampling S points from n points is $O(n)$, if we avoid duplicates. The overhead of sorting is $O(S * logS)$. Replacing S with $k * n$, the sorting overhead is $O(k*n*logn)$, where $k < 1$. If this splitter is used for L levels, the cost of tree building is $O(L * k * n * logn) + O(n)$. Since the construction is hierarchical and uses geometry, we may be able to reduce this cost by using constant number of samples.

4. Exact Median by Selection: Out of all median finding algorithms, this has been the best implementation so far. This algorithm computes median by repeatedly selecting and pruning the sample set. The cost of sampling k points from a set of N points where $k < N$ is linear in the number of points. Once a sample set is picked, and a random point is selected as the median, the rank r of this median in the sample set is computed. If the number of points less than the median is greater than half, then the correct median should have a rank less than r. If the number of points larger than the median is more than half, then the median should have a rank greater than r. The samples are pruned in either case to remove points that need not be considered during the next iteration [71]. If points are random, sampling converges quickly in constant number of steps. The cost of selecting the median from samples is linear in the number of samples. The computation cost is $O(c * N)$ where c is the number of iterations until the correct median is found. The algorithm terminates when the sample list is small enough to be sorted (exact median for a small list of size < 100).

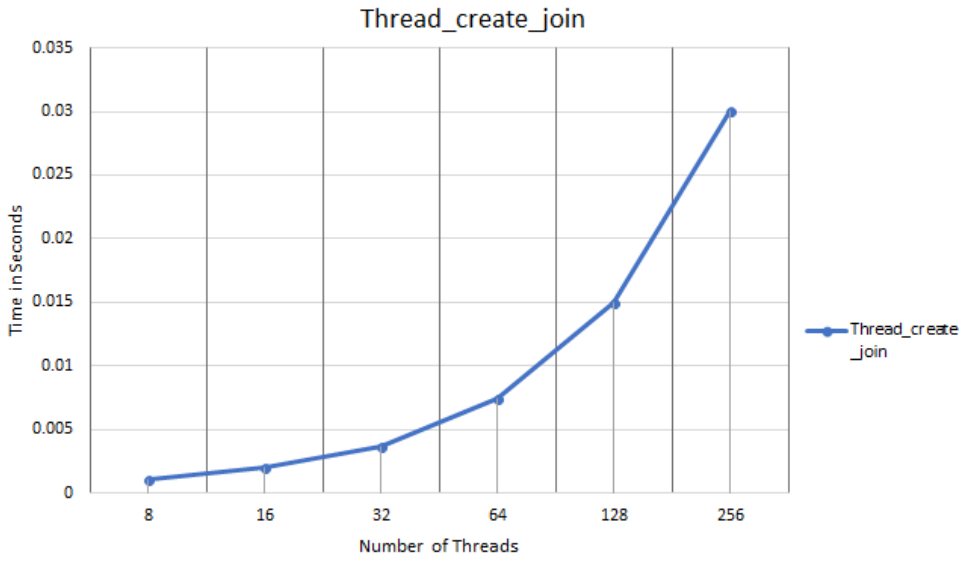


Figure 2.4: Pthreads overheads

2.4.1 Shared Memory Implementation - Threads and Synchronization

STL interface to pthreads was used for shared memory implementations in this thesis [72], [73]. The interface is minimal, which worked out best for us. We did not compare our implementation to similar programs written using contemporary threading libraries like [74], [75]. The objective was to measure the impact of data re-ordering between and within nodes. The threading interface had to be as simple as possible without any frills, for better control over data placement and mapping of threads vs data location. In this discussion, mapping threads is same as pinning them to CPUs. We use these terms interchangeably, unless mentioned. STL interface has routines to spawn threads and assign tasks to them. The synchronization constructs supported by STL are the following :

1. atomic : atomic variables and instructions like fetch-add, compare-swap used for lock free data structures [72], [76].
2. mutex variables for implementing locks [72], [76].
3. Signal and wait constructs for handling dependencies in control flow between

threads [72], [76].

This interface was used to spawn threads, assign functions to them and wait for threads to join the main thread after execution, referred to as fork-join model [77]. The assignment of tasks to threads and mapping to cores was done manually for better control. We used pthread affinity functions for mapping threads to the CPUs on a node. Tasks are functions in our implementation, which have access to shared and local data structures. All threads in a group may execute the same function with partitioned data [78] or different functions that synchronize using read/write variables or messages. If threads execute different functions, their control flow needs to be synchronized using signal-wait constructs. Some threads may block or yield while waiting for others to complete. It can also happen that there exists a dependency between any two threads t_i and t_j, e.g t_i produces a value that is required by t_j. Then t_i notifies t_j when its work is completed while t_j blocks until the signal is received. This model is supported by STL to co-ordinate execution of threads while satisfying dependencies. In the extreme case, this can be used to implement a barrier where a group of threads block until a signal is received.

We implemented a few additional constructs for brevity in our programs. A rule of thumb throughout the implementation was to keep synchronization to a minimum. Three basic synchronization constructs were identified as minimum requirements for all our programs. An array of atomic variables, one for each thread, was used to implement these functions. We call this array $wait$ [76].

1. Barrier - The members of the $wait$ array are initialized to zero. When thread t_i arrives at the barrier, it sets $wait[i] = 1$. Thread0 computes the sum of all wait variables. When the sum equals the number of threads, it resets all wait variables. The remaining threads block until their wait variables are 0. We have referred to this function as $sync$ in our programs. A variant of $sync$, known as $sync(tid, fn)$ is used to implement atomic sections of the code where tid is the thread id and fn is the atomic function. Signal and wait are implemented using loads and stores to the $wait$ array and polling the values of these variables.

2. Reduction - This primitive can be defined on any binary operation. It computes a reduction on thread local values and broadcasts the cumulative value to all threads. We have used this method for computing the sum, maximum and minimum of thread local values. We have also used it to broadcast values across threads. This method was used in some sections of the program for agree-

ment/consensus between threads. There is a shared vector called $reduce$, with $num\_threads$ values. Each thread writes to or reads from the location $reduce[tid]$, where tid is the thread id.

3. Scan - This primitive is used for quick computation of offsets and prefixes in programs. It computes an inclusive scan of all thread-local values and shares the result with the corresponding thread. This can be defined on any binary operation. There is a shared vector called $prefix$ with $num\_threads$ values. Each thread writes it local value to $prefix[tid]$ where tid is the id of a thread. For any thread t_i where $tid = i$, this function applies the binary operation bin on all values in $prefix$ starting from 0 to i. The result is stored in $prefix[i]$ and read by t_i at the end of the computation.

The complexity of these three synchronization primitives is linear in the number of threads. The optimal is $O(logP)$ where P is the number of threads [79]. But, we kept the implementation simple, because synchronization is restricted to threads in a group and this is usually a small pre-defined value. The time taken for thread creation, reduction, prefix and barrier are shown in figures 2.4,2.5. These measurements were taken on Intel KNL nodes [80]. All associated arrays were allocated on DDR memory.

All programs discussed in this thesis follow a common model, where threads are spawned once, and joined when the program terminates. This reduces the overhead of spawning threads frequently. Any intermediate synchronization is achieved using barriers (sync). There are no primitives for work distribution or load balancing between threads [81]. Work assignment to threads was done by assigning indices of the data structure to threads according to some permutation, usually in the order of thread ids. This works out particularly well for array-type data structures. We have transformed non-array type data structures to use indices for this reason. The transformations will become clearer in the later sections. If the application requires explicit mapping of threads to cores, a mapping function is defined to create different permutations. Some objectives to consider while creating these permutations is the number of available cores and the location of NUMA regions on a node. For sections of the code where there is very little data re-use for threads, scheduling is off-loaded to the operating system. For sections where data is re-used, threads are pinned to CPUs. An example where this decision would make a difference is divide-and-conquer or recursion. Implementations of such algorithms where the problem is decomposed into smaller non-overlapping sub-problems would benefit from mapping or pinning threads to CPUs.

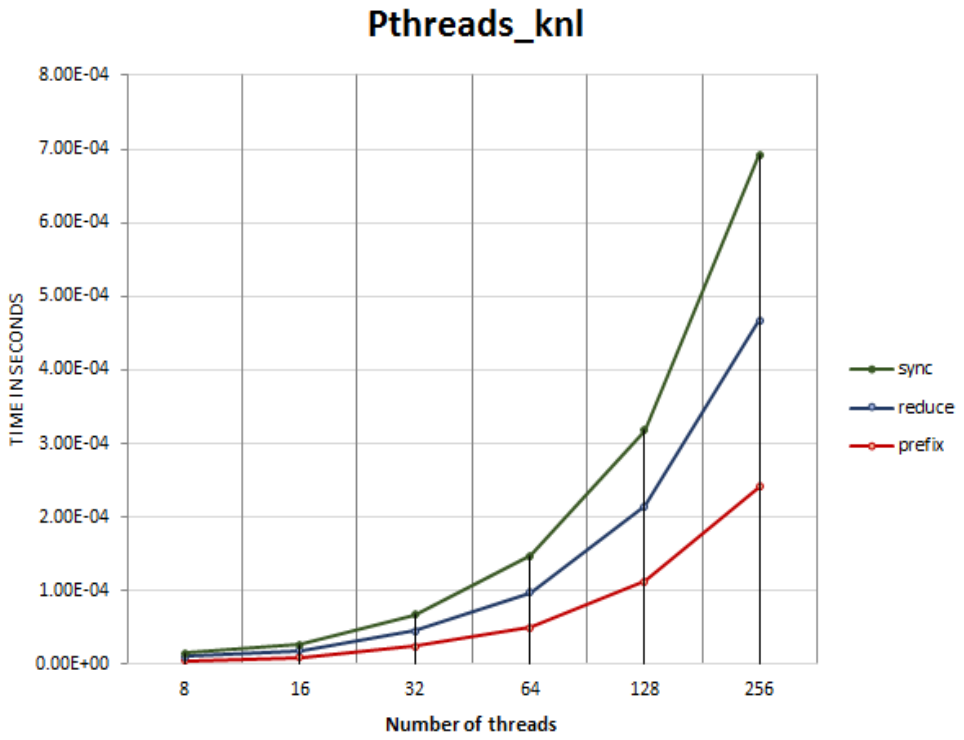


Figure 2.5: Pthread Additional Constructs

The sub-problems are more likely to be executed by the same thread-CPU combination which could benefit from cache re-use [82] for reasonable problem sizes. Pinning of threads to CPUs follows KNL numbering of tiles and cores, for a given configuration. Threads executing on CPUs that share a lower level cache, were assigned nearby data in memory, to take advantage of spatial locality. Benefits depend on the available data locality in the computation as well as the implementation.

2.4.2 Parallel KD-tree - Data structure and Algorithms

Several changes had to be made to the sequential data structure to make it parallel friendly.

1. Linear Indexing

   Parallel implementation of kd-trees discussed in this thesis are transformed linear data structures which enable quick mapping to threads and load balancing across nodes. The active nodes of the tree are numbered according to some traversal

order or global indexing scheme. In our implementations, the default order is Morton, unless specified. Hilbert-like orders are used for better locality, benefits of which are discussed in the next chapter. For now, the indexing scheme is defined as function f that maps the members of a spatial data structure with some absolute or relative distance metric defined on them in R^d to $[0, N-1]$ where the member at the origin is assigned the index 0 and the farthest member has index $N-1$. Trees are natural in this regard, because they form hierarchies and distances that are easy to compute. A simple depth-first search or breadth-first search can assign distances to the nodes of a tree, which can be translated to indices that span a global range. Some mappings are better than others :

(a) Absolute indexing: The indices assigned to the members of a data structure may be absolute, most often derived from spatial co-ordinates. For example, the elements of a mesh in 3D have (x, y, z) co-ordinates and the bit-wise interleaving of these co-ordinates can be used to generate unique keys. Any hashing function H can be defined on these co-ordinates without considering the number and position of other members in the data-structure. Although it is possible to derive unique hash keys without collisions in this scheme, the span can be very large. For 64-bit co-ordinates, bit-interleaving will generate hash keys that are $64 * 3 = 192$ bits wide. The span is therefore $[0 - (2^{192} - 1)]$. Many of these indices are likely to be unused. Managing such a large range of keys with missing values, leads to overheads that in-turn defeat the purpose of a linear compressed data structure.

(b) Relative indexing: Indices can be derived based on their relative location to other members in the data-structure. One must be careful to avoid collisions in this situation, but the span can be reduced considerably. We have used this for trees, arrays and linked-lists. It may be possible to find such transformations for other data-structures.

Linear transformation and hash keys is a natural way to obtain parallelism in data structures. Concurrent accesses to a data-structure can be made mutually exclusive by accessing members with different indices. Any permutation, Morton, Hilbert or Gensfc can be used to linearize the same data structure. In places where this is not possible, we have used atomic variables to synchronize concurrent reads and writes. Our sequential implementation also had linear trees, but it was mainly meant for internode load balancing and neighbor look-up. Keys and

indices are used to assign exclusive addresses to threads and reduce contention.

2. Buffered Writes

   Contention in programs can be reduced by using additional buffers which store partial results. We have used this technique in most of our implementations to reduce time spent synchronizing memory accesses. The idea is to maintain thread-local or shadow copies of data and accumulate partial results incrementally, using reduction and prefix methods. This technique will work provided the extra buffers are small compared to the size of the data structure. In the best case, buffer size is equal to the number of threads, because each thread has at most one value to share with others. In the worst case, the auxiliary buffer is as big as the original data structure. Locks may be a better option for such scenarios. Like in the serial version, each kd-tree node covers a sub-region in the domain bounded by axis-parallel planes. The indices of points within a sub-region are stored in contiguous memory locations. Tree nodes store the offset in the index array and number of points in their bounding boxes. The linear implementation of a kd-tree with 7 nodes is shown in the figure 2.6. 3 nodes are non-terminal and 4 are terminal nodes. The nodes are assigned monotonically increasing indices, i.e. the children of a node are assigned numbers strictly greater than itself. The root node covers the entire index array, with $offset = 0$ and $num\_points = N$. Indices are relocated in the array whenever a node is split. The indices of points to the left of the splitter are relocated to the first half and the remaining indices to the second half. To generalize, when a node with $offset = t$ and $num\_points = n$ is split into two sub cells $node1$ and $node2$ with number of points $n1$ and $n2$ respectively, the lower cell ($node1$) will have $offset = t$, $num\_points = n1$ and the upper cells ($node2$) will have $offset = t + n1$ and $num\_points = n2$.

3. Replicated Computation

   In some scenarios if data is available to all threads, it would benefit if all threads replicate some computation. This can help only if the replicated portion has fewer synchronization steps than the code section using multiple locks, barriers and broadcasts. We have employed this technique in several places in the code. For example, suppose the average of a vector needs to be computed in parallel for a small subset of data n, where n is comparable to p. It might be beneficial if all threads in the group replicate average computation without waiting for other threads to compute their local values, followed by a reduction. The cost of repli-

cated average is $O(n)$, while the cost of parallel average is $O(\frac{n}{p}) + O(p)$. For small problem sizes, synchronization costs can dominate computation cost. In such situations, replication is preferred. Our overheads are low, compared to implementations that do task scheduling. We have no auxiliary data structures for scheduling thread execution. A light-weight implementation suited this problem, but there may be other problems which require complex control structures and algorithms for thread scheduling.

4. Reduced Working Set

   The diagram in figure 2.6 shows the two-level partitioning defined on the reduced set. The mesh data or database is untouched during tree-building and partitioning.

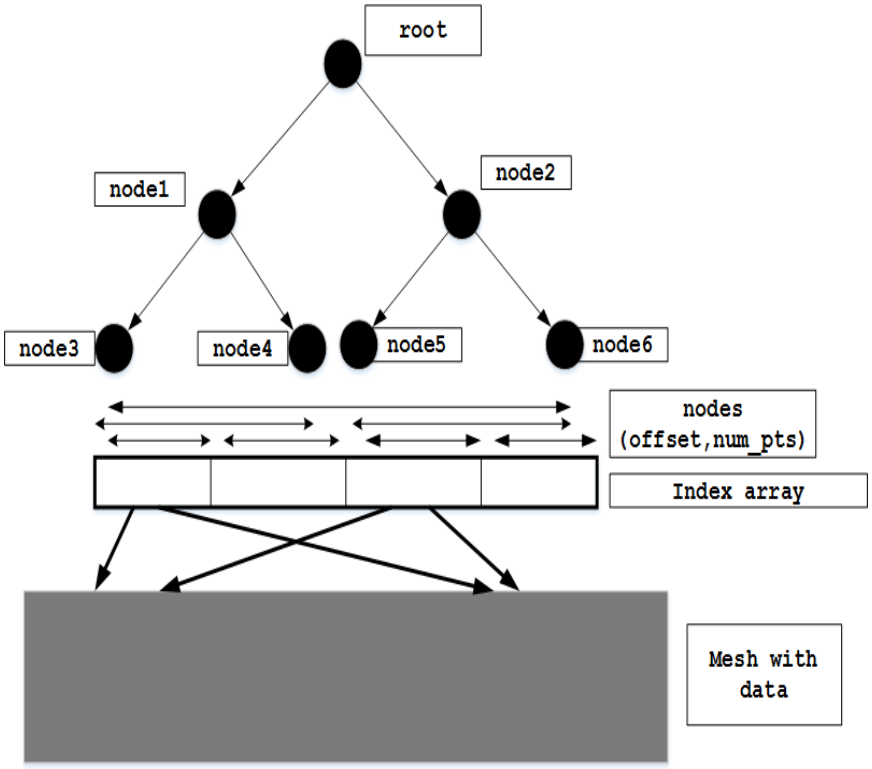


Figure 2.6: Linearized kd-tree

Top nodes of the tree which contain larger number of points, are shared by all threads. The node parameters computed during this stage are aggregated from thread-local partial values. The synchronization required for this stage is equal to the cost of reduction and prefix, which is proportional to the number of threads. The initial recursion is stopped when there are sufficient nodes (>=number of threads) to distribute across threads in a load balanced manner. After the initial phase, threads work on their

independent sets of nodes. Very little synchronization is required after this point. Tree nodes are stored in a linked-list, where the elements of the linked list are blocks of nodes. The number of nodes in a block is configurable and defined by the parameter $BLOCKSIZE$. Nodes are indexed using their unique ids. A node with id n belongs to block $\frac{n}{BLOCKSIZE}$. The index of the node within the block is $n\%BLOCKSIZE$. These two indices are used by threads to locate the node. Since node ids are global, when new nodes are added to the tree, it may lead to addition of blocks to the linked-list if its id is out of range. The addition of new blocks to the list is synchronized using atomic compare-swap operations. For sufficiently large $BLOCKSIZE$ this may happen very few times during tree-building. Also, the addition of blocks is distributed across all threads if the top node assignment is load balanced. If there are b blocks in the linked-list, location of a node belonging to some block i, $i >= 0, i < b$, will require at most $\lceil(\frac{b}{BLOCKSIZE})\rceil$ accesses to pointers in the linked-list.

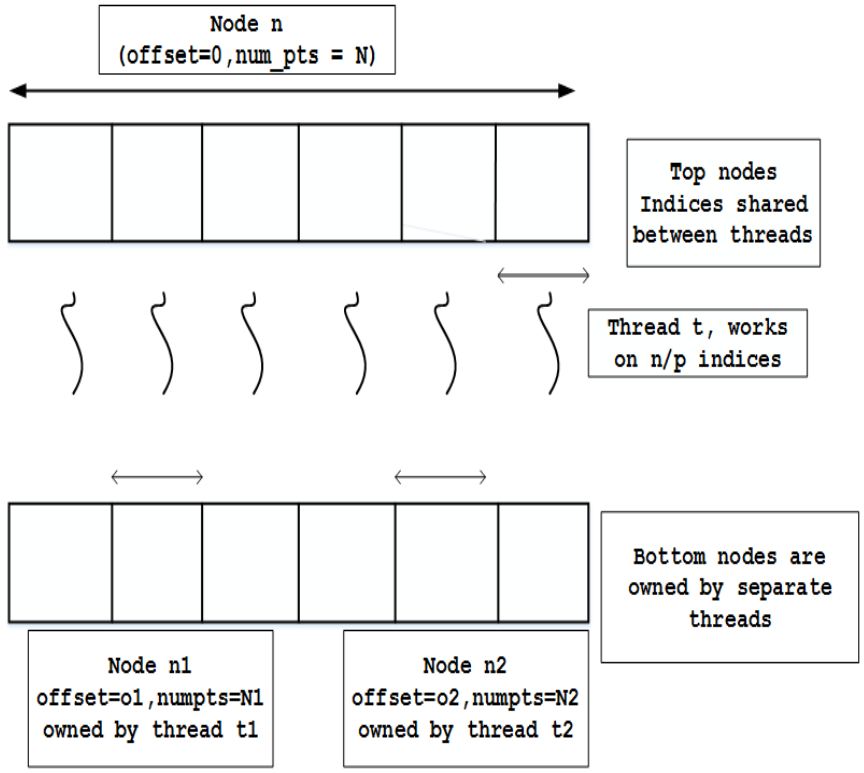


Figure 2.7: Parallel Construction of KD-trees

The program computes splitters and performs data re-ordering. Midpoint splitter has linear computation cost $O(n)$ for a node containing n points. If a node is shared by p threads, each thread computes the minimum and maximum co-ordinates of a set of $\lceil\frac{n}{p}\rceil$ points in d dimensions. The global minimum and maximum are computed from

partial results using vector reductions, each taking linear time $O(p)$.

We implemented parallel versions of all sequential median finding algorithms explained in the previous section. Pseudocodes for parallel splitters are provided in algorithms 2.4, 2.5 and 2.6. Algorithm 2.4 is the parallel midpoint finding method and algorithms 2.5 and 2.6 are the median finding methods, approximate median and median by selection resepctively. Sequential quicksort is replaced by a shared memory version of parallel quicksort discussed in section 2.10. If T_{qs} is the cost of parallel quicksort for a vector of numbers, the total pre-sorting cost is $d * T_{qs}$ for d dimensions. Exact median is computed in constant time by all threads independently without additional synchronization. This computation is replicated on threads to avoid synchronization. But, pre-sorting overheads still hold for large datasets. For the approximate median splitter, we used parallel sampling, followed by sorting. This version had similar tree building time compared to exact median. Median selection algorithm had the best parallel performance. For a set with n co-ordinates, threads select unique samples (avoid same values) independently from their subsets by traversing them once. These samples are combined to form a larger sample set. The cost of selecting S samples from a set of n points is $O(\frac{n}{p})$, where each thread selects $\lceil \frac{S}{p} \rceil$ local samples. A prefix operation is used to compute the position of samples in the sample set. The total cost of sampling is therefore $O(\frac{n}{p}) + O(p)$

Algorithm 2.6 iteratively computes the median in linear time by selection from a smaller subset. A subset of co-ordinate values are picked to form the sample set. At iteration i, sample s in the subset is either *valid* or *invalid* depending on whether it is considered in round i or not. Initially, all samples are valid. Median is selected from a set of *valid* thread-local choices. Once a median is picked, its rank in the array is computed using reduction and parallel prefix. Depending on the rank of the median, the sample set is pruned by parallel marking those samples that are out of range as *invalid*. When the number of *valid* samples in the set falls below a threshold, the entire algorithm becomes sequential. Thread0 sorts local samples and picks the middle value. During the selection phase, threads pick at most one *valid* candidate median each and share its value. These p medians are sorted, and middle value is picked. For a sample set of size S, cost of the selection phase is $O(\frac{S}{p} + plogp)$. If there are c selection steps, the total cost of selecting a median is $O(c * \frac{S}{p} + c * p * logp)$ which is at most $O(S)$, linear in the size of the sample set. Communication cost is $O(k * p)$, for some $k > 1$, if there are at most k synchronizations in c iterations.

Median splitters produce trees that are balanced with shorter maximum distance from root to leaf, especially for higher dimensional data. The maximum depth of the kd-tree depends on the distribution of points in the domain. If the points are uniformly distributed in the domain, there may not be a big difference between median and midpoint values in a dimension. For highly clustered point sets, median splitters produce shorter trees that reduce both tree building time and computation time for operations performed on data stored in the tree, such as searching [65]. For clustered data, we have used a combination of splitters, where median is used at the top nodes and midpoint at the lower nodes of the tree.

The implementation is divided into two versions, to suit the nature of input data. It is easier to manage memory and data structures for static datasets, compared to dynamic inputs which move or change in size (points may be added or deleted). There are two multi-threaded implementations which are described in detail below:

2.5 STATIC KD-TREE

Algorithm 2.3 lockfree Nodelist

```
1:  procedure Add_new_node(curmax)
2:      b ← ALLOCATE_NEW
3:      do
4:      n ← 0
5:      a ← memory_blocks
6:      While(a− > next.load() ≠ NULL)
7:      a ← a− > next.load()
8:      n ← n + 1
9:      if then curmax ≤ n ∗ BLOCKSIZE
10:         break
11:     end if
12:     !While(a− > next.cas(a− > next.load(), b))
13: end procedure
```

A pseudo-code for addition of new blocks is provided in the algorithm 2.3. In the current implementation of static kd-trees, blocks are never deleted. The tree once constructed, is maintained in its entirety until the program terminates. For static datasets, it may be beneficial in some cases to avoid storing non-terminal tree nodes in the node list, to reduce memory footprint. A vector of leaf nodes may be sufficient for most problems. However, we have maintained the non-terminal nodes in our implementa-

Algorithm 2.4 Parallel_Midpoint_Splitter

```
1:  procedure Par_Midpoint(node *n)
2:      offset ← GET_OFFSET(n)
3:      numpts ← GET_NUM_POINTS(n)
4:      avg_load ← numpts / num_threads
5:      tid ← THREAD_ID
6:      offset ← offset + avg_load * tid
7:      N ← GET_NUM_POINTS(n)
8:      for do i = offset, offset + N/p
9:          for do j = 1, NDIM
10:             if then min_t[j] > data[i].coord[j]
11:                 min_t[j] ← data[i].coord[j]
12:             end if
13:             if then max_t[j] < data[i].coord[j]
14:                 max_t[j] ← data[i].coord[j]
15:             end if
16:         end for
17:     end for
18:     THREAD_BARRIER
19:     for do j = 1, NDIM
20:         min[j] ← THREAD_REDUCE(min_t, MINIMUM)
21:         max[j] ← THREAD_REDUCE(max_t, MAXIMUM)
22:     end for
23: end procedure
```

Algorithm 2.5 Parallel_Approx_Median_Splitter

```
1:  procedure Par_Approx_Median(node *n)
2:      offset ← GET_OFFSET(n)
3:      numpts ← GET_NUM_POINTS(n)
4:      numsamples ← NUM_RANDOM_SAMPLES
5:      samples ← PICK_SAMPLES
6:      THREAD_BARRIER
7:      PARALLEL_SORT(samples)
8:      THREAD_BARRIER
9:      median ← samples[(numsamples/2)]
10: end procedure
```

Algorithm 2.6 Parallel_Median_Selection

```
1:  procedure Par_Approx_Median(node *n)
2:      offset ← GET_OFFSET(n)
3:      numpts ← GET_NUM_POINTS(n)
4:      numsamples ← NUM_RANDOM_SAMPLES(numpts)
5:      tid ← threadID
6:      avg ← ⌈numsamples / numthreads⌉
7:      samples ← PICK_SAMPLES
8:      start ← tid * avg
9:      end ← start + avg
10:     num_less_g ← 0
11:     num_grt_g ← 0
12:     THREAD_BARRIER
13:     While(true)
14:     med_local ← PICK_VALUE(start, end)
15:     med_samples[tid] ← med_local
16:     THREAD_BARRIER
17:     SEQUENTIAL_SORT(med_samples)
18:     med ← PICK_MEDIAN(med_samples)
19:     num_less ← COUNT_LESS(med)
20:     num_grt ← COUNT_GRT(med)
21:     num_less_g ← num_less_g + THREAD_REDUCE(num_less, SUM)
22:     num_grt_g ← num_grt_g + THREAD_REDUCE(num_grt, SUM)
23:     if then abs(num_less_g − num_grt_g) < small
24:         median ← SEQUENTIAL_SORT_MED(sample) return median
25:     else if then num_less_g > num_grt_g
26:         MARK_RIGHT(samples)
27:     else
28:         MARK_LEFT(samples)
29:     end if
30:     EndWhile
31: end procedure
```

tions of both static and dynamic kdtrees. We certainly need the entire tree for dynamic versions which require changes to the data structure, i.e addition of new leaves and deletion of existing leaves. When the number of points covered by a node falls below bucket size, the node is marked as a leaf and tree-building ends for that sub-cell. When threads are simultaneously working on independent sub cells, the total execution time depends on the longest path, total number of nodes and the largest sub-cell assigned to any thread. Therefore, to reduce parallel execution time, it is important to ensure load balance across threads. One of the ways to do that is to create sub cells that have nearly equal number of points. The other option is to partition the tree until there are $r * num\_threads$ non-terminals in the tree, where $r >= 1$. Weights are assigned to nodes, where weight of a node is equal to the number of points or a normalized fraction of the total number of points. If computation is unequal, one can define a distribution of weights, normalized over the maximum value at any point. All threads are assigned roughly equal weights of sub cells. This will ensure they have similar load during the second stage of parallel tree building. For the shared memory tree implementations, we have created sub cells with roughly equal number of points using splitters that balance the load. Weighted load balancing is used for assigning subcells to processes in the distributed version. Although we handle input data as points throughout this discussion, these results hold both for structured grids and for unstructured meshes.

Parallel splitters partition data according to the value and dimension of the splitting hyperplane. The pseudocode for parallel splitting is described in algorithm 2.7. The splitting hyperplane may be computed by parallel $midpoint$ or $median$ functions. Suppose $dim = i$ and $s\_value = s$, this algorithm rearranges data into two subsets $<= s$ and $> s$, along dimension i. The parallel splitter rearranges data indices concurrently. If $left$ is the total number of points $<= s$ along dimension i, data is rearranged by computing offsets of these points in the index vector. If $offset$ is the first location in the index array belonging to the node, all points $<= s$ will have offsets computed relative to $offset$, while points $> s$ will have offsets relative to $offset + left$. Suppose the splitter uses midpoint values to separate data, and T_S is the time taken for splitting along the midpoint. Given midpoint computation is linear (including computing the boundaries of boxes) in the number of points per thread, T_S is :

$$T_S = O(\frac{n}{p}) + O(p) \tag{2.7}$$

Algorithm 2.7 Parallel_Splitter

1: **procedure** $Par\_Splitter$(node *n)
2: PAR_MIDPOINT(n)
3: $dim \leftarrow$ SPLIT_DIM(n)
4: $s\_value \leftarrow$ SPLIT_VALUE(n)
5: $offset \leftarrow$ GET_OFFSET(n)
6: $N \leftarrow$ GET_NUM_POINTS(n)
7: $Nz \leftarrow$ NUM_ELEMENTS
8: $tid \leftarrow$ THREAD_ID
9: $avg\_load \leftarrow \frac{N}{P}$
10: $offset \leftarrow offset + tid * avg\_load$
11: **for do**$i = offset, offset + \frac{N}{p}$
12: **if then**$data[i].coord[dim] \leq s\_value$
13: $left\_t \leftarrow left\_t + 1$
14: **else**
15: $right\_t \leftarrow right\_t + 1$
16: **end if**
17: **end for**
18: $l\_id \leftarrow cur\_node.fetch\_add$
19: $u\_id \leftarrow cur\_node.fetch\_add$
20: SET_LOWER(n, l_id)
21: SET_UPPER(n, u_id)
22: $left \leftarrow thread\_reduce(left\_t, SUM)$
23: $right \leftarrow thread\_reduce(right\_t, SUM)$
24: COPY_BUFFER($offset + Nz, left\_t$)
25: COPY_BUFFER($offset + Nz + left, right\_t$)
26: THREAD_BARRIER
27: RELOCATE($offset, left\_t$)
28: RELOCATE($offset + left, right\_t$)
29: **if then**$tid == 0$
30: SET_OFFSET($l\_id, offset$)
31: SET_OFFSET($u\_id, offset + left$)
32: SET_NUM_POINTS($l\_id, left$)
33: SET_NUM_POINTS($u\_id, right$)
34: **end if**
35: $thread\_barrier$
36: **end procedure**

T_S will have the same overheads if using a median selection algorithm. If using other functions to split data, T_S should be replaced accordingly.

For approximate median, the cost is :

$$T_S = O(\frac{n}{p}) + O(\alpha * \frac{n}{p} * log(\frac{n}{p})) + O(p) \tag{2.8}$$

This is the algorithm that uses a smaller sorted subset $S = \alpha * n$, where $\alpha < 1$ of samples to approximate the median.

T_S is therefore the cost of splitting nodes at depth i, where $0 <= i < log(K)$, if there are K top nodes. This is the shared portion of the tree where decisions are taken collaboratively. Therefore the cost of building the top levels of the tree, T_{tp}, is :

$$T_{tp} = (logK) * T_S \tag{2.9}$$

The lower levels of the tree are constructed independently in parallel by p threads. For $r = 1$, assume top nodes are distributed roughly equally between threads, with each thread having $\lceil \frac{n}{p} \rceil$ points on an average in its subcells. If the size of a bucket is at most b points, the average number of leaves in a subtree with $\lceil \frac{n}{p} \rceil$ points is $\lceil \frac{n}{p*b} \rceil$, each leaf node containing b points.

$$T_{ip} = O(log(\frac{n}{pb}) * \frac{n}{p}) \tag{2.10}$$

In our results, these execution times are divided into two columns $init\_time$ and $build\_time$. T_{tp} is the $init\_time$ and T_{ip} is the $build\_time$. The differences between tree-building times with different splitters can be justified by plugging in their respective cost functions and comparing with measured values.

2.6 TESTCASES

Two kinds of testcases were used for evaluating the static kd-tree and the splitters separately. When points are distributed uniformly in the domain, there is no difference between the midpoint and median values. The cheaper splitter is the better op-

tion, which is usually midpoint. Input datasets were built by sampling points from a uniform distribution [83]. Each experiment uses a different uniform distribution within the same range. Therefore, execution times may differ across runs. The values reported are averaged over five runs, for different thread counts and problem sizes. The reported execution times are for tree building alone without the overhead of SFC traversal. Traversal times are reported in the next chapter. The second case is used to evaluate three median finding algorithms. A clustered distribution is used for this to separate midpoint kdtrees from median trees. The dataset was built from an underlying uniform distribution with one cluster located in the bottom left corner. The cluster was generated using a Poisson distribution [83] with median in the lower left corner of the domain. This combination of distributions was used to generate a pointset with boundaries fixed by the programmer.

The results are tabulated below.

#threads	num_points	num_nodes	bsize	max_depth	init_time	build_time
64	1m_3D	88437	32	18	0.278551	0.194691
128	1m_3D	88437	32	18	0.436313	0.25911
256	1m_3D	88437	32	18	0.916542	0.389991
64	1m_10D	88595	32	17	0.45169	0.209525
128	1m_10D	88595	32	17	0.800992	0.266701
256	1m_10D	88595	32	17	1.71579	0.415911
64	10m_3D	931055	32	22	0.859855	2.82839
128	10m_3D	931055	32	22	0.957406	3.76726
256	10m_3D	931055	32	22	1.42395	2.87436
64	10m_10D	940055	32	21	0.987128	13.7839
128	10m_10D	940055	32	21	1.40502	13.2486
256	10m_10D	940055	32	21	2.86587	6.41648
64	100m_3D	2168975	128	23	7.33959	57.8845
128	100m_3D	2168975	128	23	5.76166	48.0373
256	100m_3D	2168975	128	23	6.017	29.9769

Table 2.1: Static KD-tree construction time, uniform dist, midpoint splitter,scheduled and pinned

The measured times for tree building, with four splitters - midpoint splitter, exact median, approximate median and median selection and two point distributions are discussed in this section. All experiments were conducted on Stampede, which is a supercomputer at TACC, Texas Center for Supercomputing [84]. Stampede has two types of nodes - Intel KNL and Intel SkyLake. All measurements in this section are from KNL nodes. We have used the default configuration - cache quadrant, with one NUMA

node [80]. The fast memory, MCDRAM is used in cache mode. The configuration on Stampede assigns the entire MCDRAM as L3 cache. The performance numbers are tabulated below. Top-level recursion terminates at **128** nodes. This number is kept constant for all experiments. The design is similar to Intel's implementation [80], although we have used arrays and indices instead of pointers. We have also implemented more splitters in our partitioner. Measurements are included for three data sizes - 1million, 10 million and 100million points and two dimensions - 3D and 10D.

#threads	num_points	num_nodes	bsize	max_depth	init_time	build_time
64	1m_3D	88367	32	17	232.826	0.133796
128	1m_3D	88367	32	17	233.806	0.103972
256	1m_3D	88367	32	17	237.594	0.116932
64	1m_10D	88425	32	17	690.08	0.185988
128	1m_10D	88425	32	17	690.469	0.219978
256	1m_10D	88425	32	17	717.584	0.364945

Table 2.2: Static KD-tree construction time, uniform dist, exact median splitter,always pinned

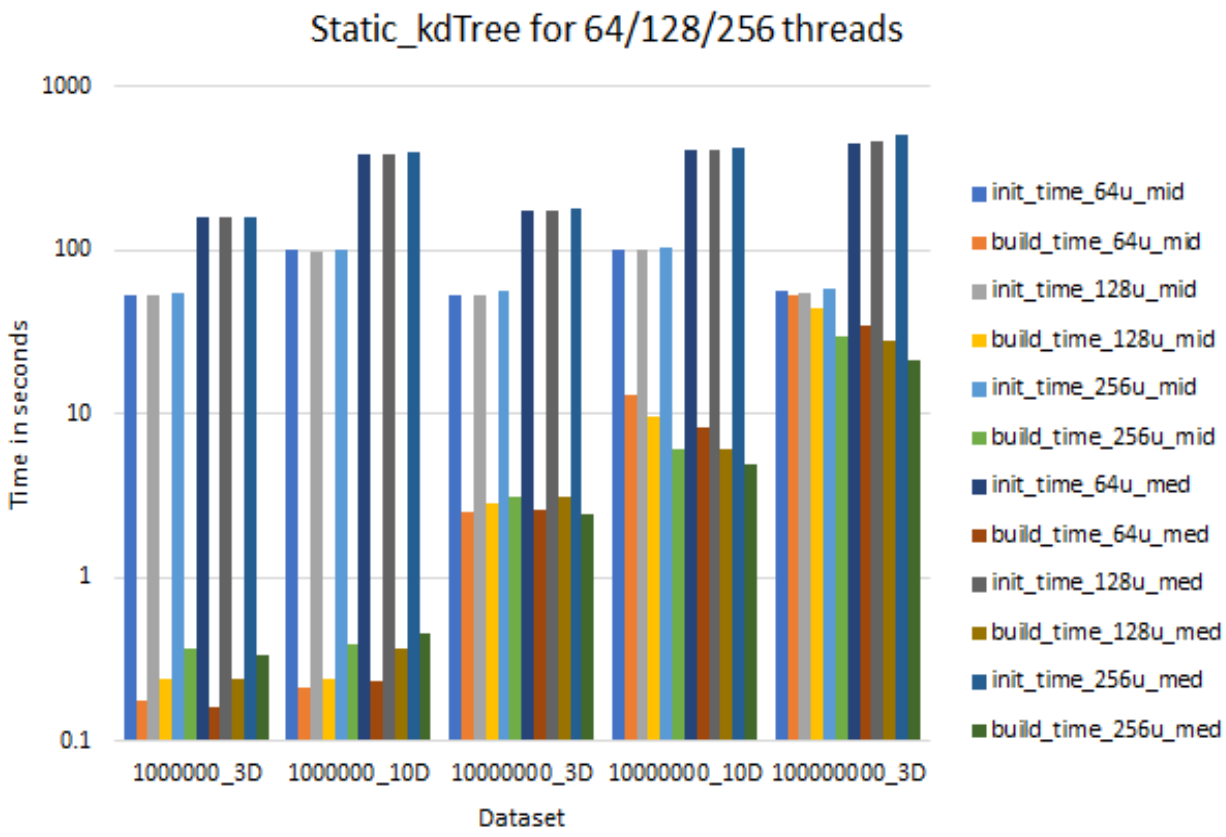


Figure 2.8: Static KD-tree, uniform, scheduled and pinned

#threads	num_points	num_nodes	bsize	max_depth	init_time	build_time
64	1m_3D	90091	32	19	94.9519	0.179216
128	1m_3D	89361	32	20	111.763	0.298883
256	1m_3D	89503	32	21	296.208	0.358957
64	1m_10D	89111	32	19	111.630	0.251025
128	1m_10D	89719	32	20	201.437	0.395938
256	1m_10D	89187	32	20	609.889	0.482809
64	10m_3D	914111	32	22	143.848	3.28883
128	10m_3D	903371	32	22	162.102	3.0292
256	10m_3D	896253	32	22	356.121	2.26295
64	10m_10D	915017	32	21	142.015	7.92856
128	10m_10D	891581	32	21	196.106	6.74519
256	10m_10D	899969	32	21	664.909	4.51095
64	100m_3D	2142645	128	23	237.056	27.571
128	100m_3D	2178319	128	23	360.967	25.472
256	100m_3D	2225813	128	23	818.57	20.462

Table 2.3: Static KD-tree construction time, uniform dist, approx.median splitter,scheduled and pinned

For both uniform and clustered data distributions, measured results are consistent with expected values. Not many experiments were performed using exact median splitter. The initial sorting routines made it an expensive splitter, especially for the data sizes used in this section. Besides, the approximate splitter gave reasonably good results, close to the exact median splitter. These results for uniformly random distributed points are tabulated in tables,table 2.1, 2.2 and table2.3. Approximate median was a good replacement for large datasets. For every splitting phase $\frac{1}{1000}$ of the number of points were used as samples. For small nodes, the entire dataset was used as samples. The computation times for the lower levels of the tree, shows good scaling with increasing number of threads, and for large datasets. Top nodes are expensive to build. Construction time depends on data size, number of dimensions and the amount of synchronization between threads. For the init phase, threads synchronize to compute node extents in all dimensions, and to pick the splitter. We have chosen different bucket sizes for each test case. Bucket sizes are mentioned in the tables. Tree building time is broken into $init\_time$ and $build\_time$ in the tables for better comparison, across datasets, dimensions and splitters. This section is an evaluation of the partitioner as a stand-alone program, without any dependence on applications. For all inputs with uniform distribution, midpoint splitter gave the best performance. $build\_time$ values were higher for midpoint splitter, for increasing data size and number of dimensions, but $init\_times$

values were much lower than other splitters. Since its performance numbers (maximum depth and $build\_time$) were very close to approximate median, the exact median splitter was dropped from remaining experiments. The median values computed using a smaller sample set were good enough to replace the exact median. The graphs for static kdtree with uniform distribution are shown in figure 2.8.

#threads	num_points	num_nodes	bsize	init_time	build_time	max_depth
64	1m_3D	89929	32	0.293638	2.22074	41
128	1m_3D	89929	32	0.450009	3.31495	41
256	1m_3D	89929	32	0.879423	4.93171	41
64	1m_10D	109609	32	0.511841	4.84628	46
128	1m_10D	109609	32	0.828893	5.91806	46
256	1m_10D	109609	32	1.78326	8.60049	46
64	10m_3D	907387	32	1.04578	47.5189	46
128	10m_3D	907387	32	1.12306	64.9748	46
256	10m_3D	907387	32	1.67447	91.5091	46
64	10m_10D	1007555	32	1.17938	168.911	57
128	10m_10D	1007555	32	1.88338	188.434	57
256	10m_10D	1007555	32	3.46121	223.096	57
64	100m_3D	2243255	128	7.70986	549.983	51
128	100m_3D	2243255	128	7.54894	760.21	51
256	100m_3D	2243255	128	8.85877	1016.77	51

Table 2.4: Static KD-tree construction time, cluster, midpoint splitter,scheduled and pinned

For the clustered distribution, tree-building times for midpoint splitter are not consistent because of higher tree depth. Nodes are not balanced and the maximum depth of the tree is quite high. Midpoint values are independent of the distribution and computed using the geometry of the domain. This lead to the creation of some nodes which had a large proportion of points compared to others. Load imbalance in subcells resulted in higher maximum execution times. This difference is apparent in the results in this section tabulated in tables 2.4 and 2.5. There is not much variance in the $init\_time$ (construction of top nodes), but the independent section is time consuming and shows poor scaling. Median splitter, has good performance especially for large data sizes (100 million points, 3D). There is no difference in the $init\_time$ for median splitter compared to uniform distribution, sorting overheads are still predominant in their total execution cost. The build times are lower than those which use midpoint splitters for large datasets. The graphs for static kdtree with clustered datasets in shown in figure 2.9.

#threads	num_points	num_nodes	bsize	init_time	build_time	max_depth
64	1m_3D	91237	32	79.2418	0.204402	35
128	1m_3D	91125	32	122.713	0.275941	36
256	1m_3D	91411	32	296.482	0.489907	36
64	1m_10D	107669	32	127.736	0.367086	39
128	1m_10D	107371	32	183.169	0.381948	41
256	1m_10D	107955	32	623.485	0.529948	40
64	10m_3D	913455	32	142.893	3.46236	41
128	10m_3D	911833	32	179.098	2.94797	40
256	10m_3D	911057	32	184.944	3.26594	41
64	10m_10D	1008465	32	418.817	11.9046	48
128	10m_10D	1008217	32	419.919	8.8217	48
256	10m_10D	1007025	32	438.738	6.38393	48
64	100m_3D	2221115	128	476.184	45.059	45
128	100m_3D	2217763	128	470.424	35.844	44
256	100m_3D	2215601	128	512.191	26.0079	44

Table 2.5: Static KD-tree construction time, cluster, approx.median splitter,scheduled and pinned

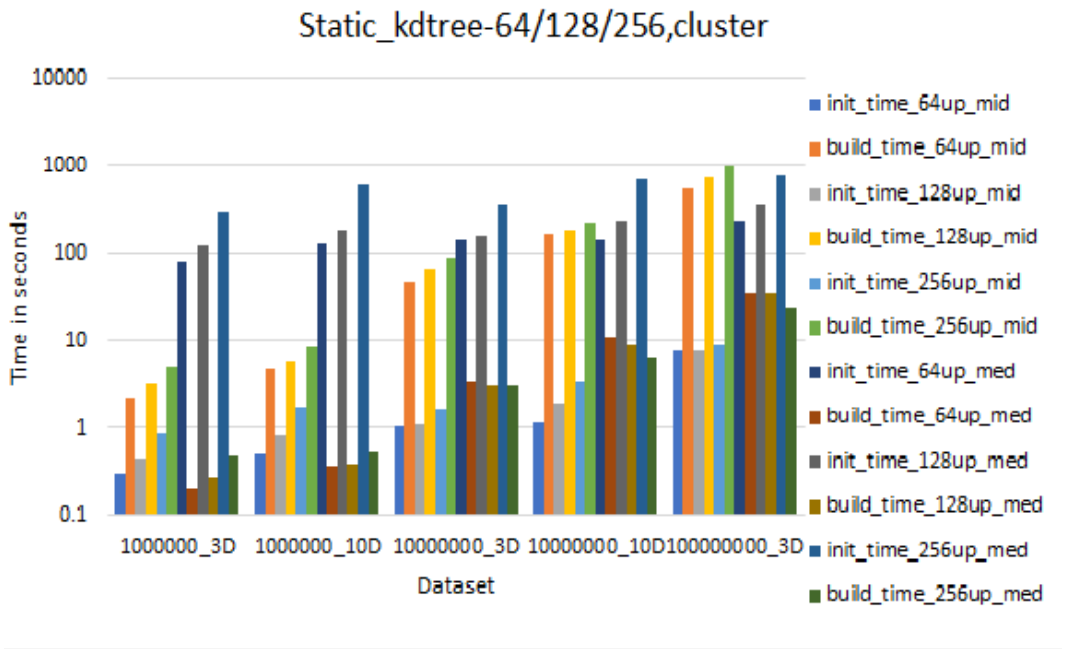


Figure 2.9: Static KD-tree, cluster, scheduled and pinned

To reduce the $init\_time$ of tree building with median splitters, the approximate median was replaced by parallel median selection algorithm. For a vector with S samples, c iterations of median selection, the total cost of parallel selection is $O(\frac{S}{p}) + O(p * logp) + O(p)$, which is linear in the number of samples for $S > p$. The improved median splitter was picked as the best performing version. The results using median selection algorithm for clustered datasets are tabulated in table 2.6

#threads	num_points	num_nodes	bsize	init_time	build_time	max_depth
64	1m_3D	90393	32	1.286881	0.189238	31
128	1m_3D	90039	32	0.796237	0.145783	31
256	1m_3D	90191	32	1.604486	0.191034	31
64	1m_10D	110949	32	0.70703	0.236332	32
128	1m_10D	111129	32	1.002143	0.193194	33
256	1m_10D	111127	32	2.014646	0.227383	32
64	10m_3D	915043	32	1.74959	4.085286	39
128	10m_3D	914925	32	1.647036	2.881918	39
256	10m_3D	915077	32	2.563302	2.875	39
64	10m_10D	1032087	32	1.446788	13.40398	44
128	10m_10D	1032133	32	1.716942	10.68796	44
256	10m_10D	1031867	32	3.075664	6.608078	44
64	100m_3D	2211459	128	12.5729	60.50094	44
128	100m_3D	2211355	128	7.592104	50.8479	44
256	100m_3D	2211491	128	6.517666	31.44724	44

Table 2.6: Static KD-tree construction time, cluster, median selection splitter,scheduled and pinned

It is left to the programmer to pick splitters depending on the distribution and cluster size. In our implementations, the observed performance was the following, decreasing in cost :

exact median > approximate median > median selection

#threads	init_time	build_time	total_time
8	28.21122	151.9588	183.653025
16	17.3582	107.08134	120.130375
32	22.07462	82.28648	104.825325
64	9.758814	71.98786	79.758045
128	6.212152	35.40592	42.642715
256	4.53673	27.07604	31.715255

Table 2.7: Static KD-tree construction time, uniform distribution,100million points

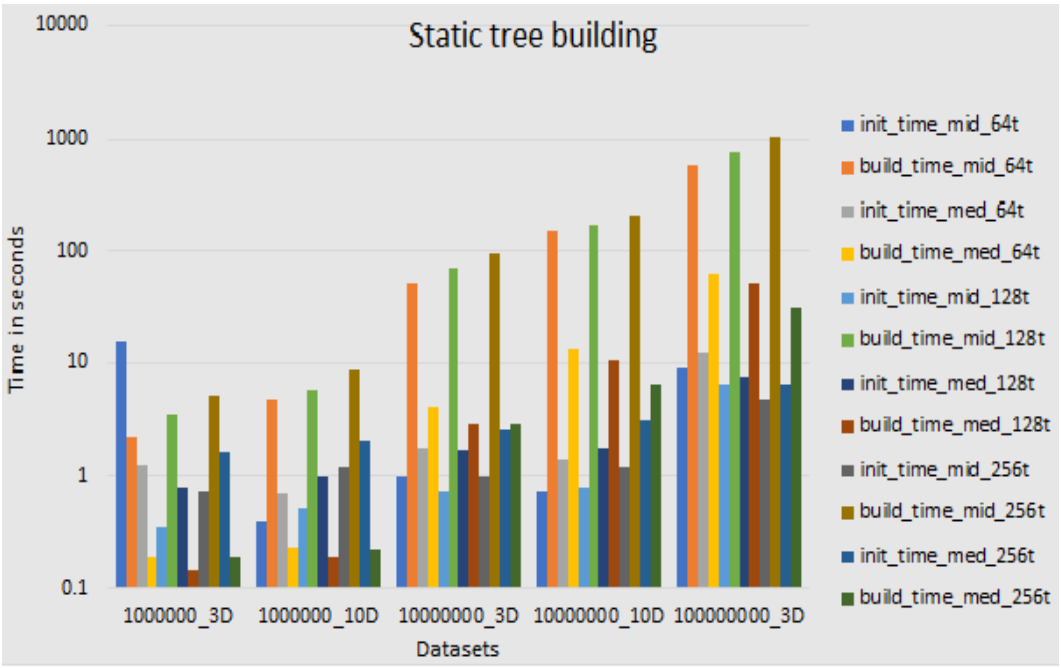


Figure 2.10: Static KD-tree, cluster, median selection, scheduled and pinned

#threads	init_time	build_time	total_time
8	2.829174	12.009516	14.83869
16	1.80409	9.56192	11.36601
32	2.307998	6.024164	8.332162
64	1.232224	4.73545	5.830922
128	1.027945	3.049212	4.077157
256	1.483622	3.626692	5.110314

Table 2.8: Static KD-tree construction time, uniform distribution,10million points

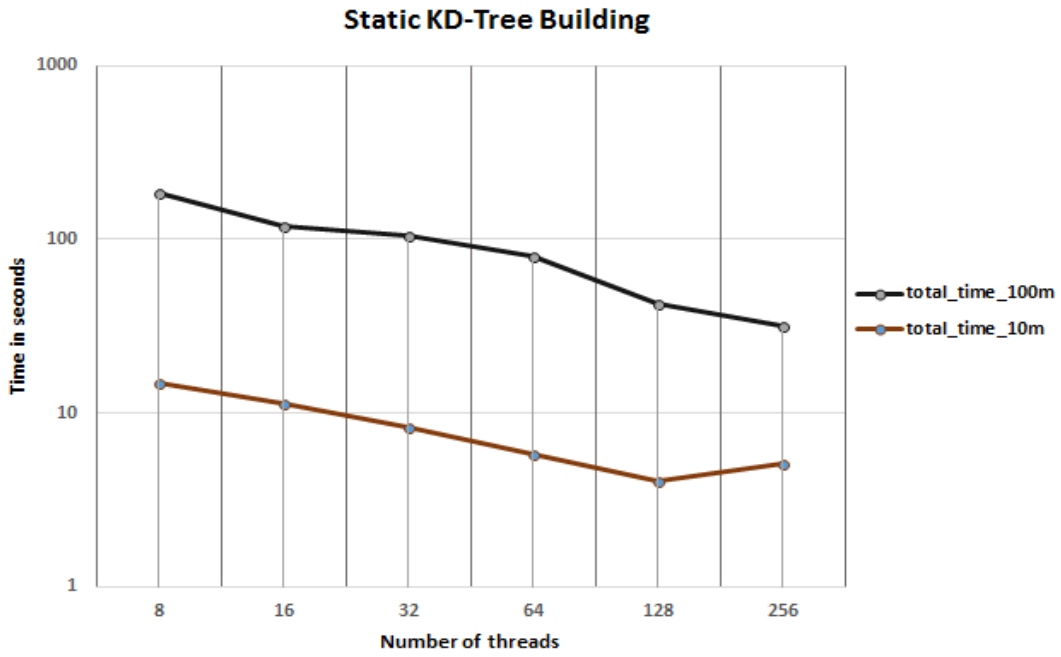


Figure 2.11: Static KD-tree, strong scaling, uniform distribution, midpoint splitter

Before closing the section on static kd-trees, we present strong scaling results for kd-tree building on a single node. The testcase used was a uniform distribution with 100million points in 3D. Midpoint splitter was used for constructing the tree. Number of threads were varied from 8 to 256. All tests were conducted on Intel KNL nodes. We used the high bandwidth memory (MCDRAM) as L3 cache. Results are tabulated in tables 2.7 and 2.8 and the corresponding graph is shown in 2.11. The values plotted on the y-axis are based on logarithmic scale. There are 34 functional tiles, 68 cores and 272 hardware threads per KNL node. Out of these resources, we have used at most 32 tiles and 256 threads. The performance of static tree building shows good scaling for number of threads ≥ 64, when there is at least one thread executing on each core.

2.7 LINEARIZING RECURSION

Tree building can be linearized by replacing recursion with while loops that index the arrays. Separate arrays are maintained for $offsets$ and number of points that lie within a node. The indices to points are stored in contiguous memory locations like in previous implementations. For the top nodes, we used specific indices for the lower and upper sub cells of a node. For example, the children of a node with index x will

have indices $2 * x + 1$ and $2 * x + 2$. We could avoid storing links between nodes with this numbering, but there were huge gaps at the lower levels of the tree where majority of leaf nodes were located. To avoid this, we used contiguous indices for tree nodes and maintained three separate arrays for storing lower, upper and parent node numbers. The diagram in figure 2.12 shows the data structure with separate linked lists for each link. For a mesh element with index a, the straight line shows all associated indices, i.e the location of its parent node id and the ids of its sub cells - lower and upper. When the number of nodes reached a threshold ($>= r * p, r >= 1$), the algorithm shifted to independent parallel execution streams, one for each thread. $offsets$, num_points,$lower$,$upper$ and $parent$ nodes were thread-local vectors for the second stage. The improvement from previous versions comes from lack of synchronization in this implementation.

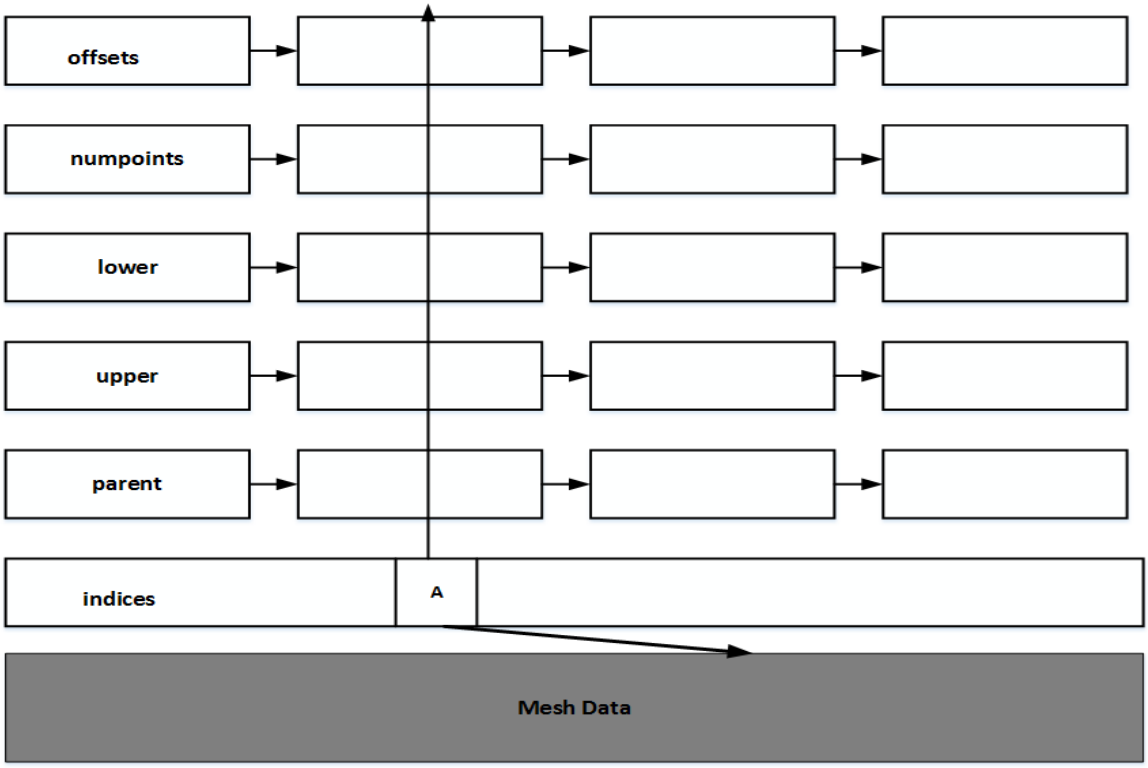


Figure 2.12: Linear Tree Data structure

The only stage which requires any synchronization is the first stage where the top nodes of the tree are built. Addition of blocks to linked-lists is done by thread0 in this stage. All data structures in the second stage are thread-local. There is no need to synchronize during addition of blocks to local lists. The algorithms for splitters are also independent computations. This was the implementation that gave the best

performance with increasing number of threads, especially 256 threads per KNL node, i.e at most 4 hardware threads per CPU. It is difficult to get good performance for this configuration because of possible thrashing in L2 caches. This was one of those implementations that scaled well.

Converting the tree building algorithm to an iterative while loop does a BFS exploration of the tree. The iterations end when there are no more non-terminal nodes to explore. When new nodes are added to the tree, their offsets and sizes are stored starting from the last index of the current iteration. The iteration indices are updated for another step. All data structures in this implementation are dynamic arrays. To avoid frequent reallocation and copying of memory, we used blocked linked-lists. The elements of a block are unsigned long integers, each 8-bytes wide. The number of integers in a block is configurable, using the $BLOCKSIZE$ parameter.

We used this version for clustered datasets, compared the tree building times for both midpoint and median selection splitters.

#threads	num_points	num_nodes	bsize	init_time	build_time	total_time
64	1m_3D	89899	32	0.3610058	1.947036	2.3080418
128	1m_3D	89899	32	0.3982278	2.267358	2.6655858
256	1m_3D	89899	32	1.205302	3.912588	5.11789
64	1m_10D	108943	32	0.5000978	4.136794	4.6368918
128	1m_10D	108943	32	0.6220068	5.401862	6.0238688
256	1m_10D	108943	32	2.385004	7.770392	10.155396
64	10m_3D	908319	32	1.190618	27.34488	28.535498
128	10m_3D	908319	32	0.865415	31.8203	32.685715
256	10m_3D	908319	32	1.534156	57.12622	58.660376
64	10m_10D	1004789	32	0.867095	58.75598	59.623075
128	10m_10D	1004789	32	0.8979384	77.10398	78.0019184
256	10m_10D	1004789	32	2.54122	112.5628	115.10402
64	100m_3D	2243267	128	10.5093	340.76	351.2693
128	100m_3D	2243267	128	6.30947	395.114	401.42347
256	100m_3D	2243267	128	5.9469	696.769	702.7159

Table 2.9: Linear KD-tree construction time, cluster, midpoint selection splitter, scheduled and pinned

The results for building linear trees are shown in figure 2.13 and observations are tabulated in tables 2.9 and 2.10. All results presented in this section for linear trees are for a clustered dataset. We have kept both implementations in our partitioner. The programmer may choose either based on his application requirements. If a simple SFC order of leaf nodes is all that is required, then linear trees may be a good option.

#threads	num_points	num_nodes	bsize	init_time	build_time	total_time
64	1m_3D	90111	32	0.698046	0.1546014	0.8526474
128	1m_3D	90061	32	0.9939328	0.07971194	1.07364474
256	1m_3D	90191	32	3.131826	0.06130304	3.19312904
64	1m_10D	110903	32	0.802467	0.149431	0.951898
128	1m_10D	110979	32	1.329398	0.10008498	1.42948298
256	1m_10D	111103	32	4.50117	0.08380948	4.58497948
64	10m_3D	914973	32	2.108454	1.778088	3.886542
128	10m_3D	914925	32	2.121596	0.8367502	2.9583462
256	10m_3D	917107	32	5.654932	0.5720838	6.2270158
64	10m_10D	1032089	32	1.827046	2.056782	3.883828
128	10m_10D	1032133	32	2.14331	1.340482	3.483792
256	10m_10D	1032357	32	6.637326	0.8479562	7.4852822
64	100m_3D	221459	128	13.35354	19.69582	33.04936
128	100m_3D	2211355	128	9.365504	10.67308	20.038584
256	100m_3D	2198161	128	12.45152	7.156178	19.607698

Table 2.10: Linear KD-tree construction time, cluster, median selection splitter,scheduled and pinned

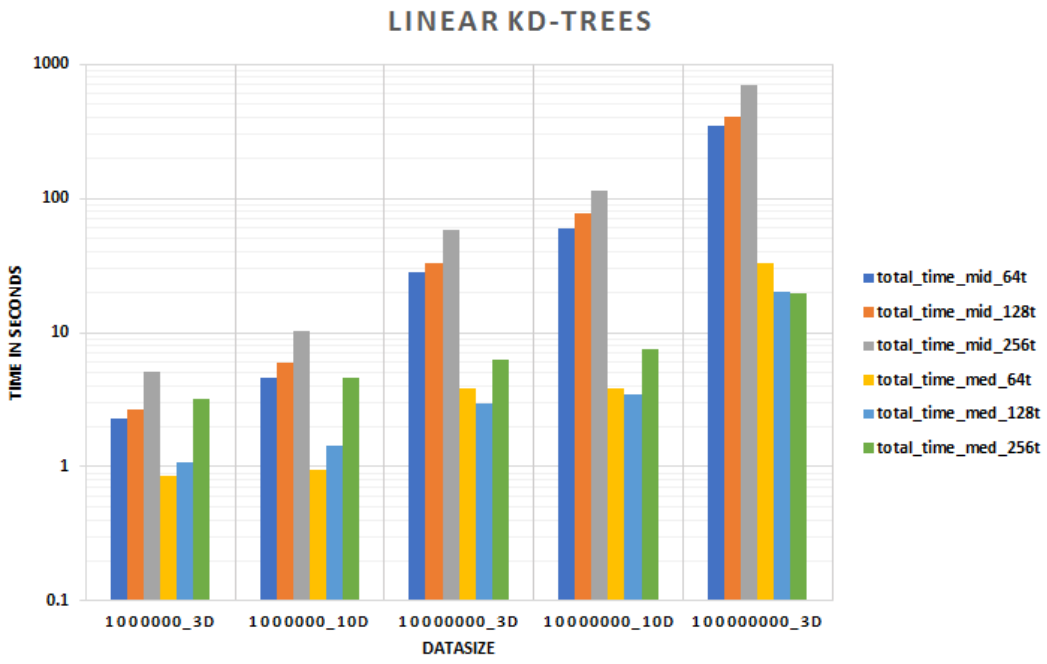


Figure 2.13: Static KD-tree, cluster, linear, scheduled and pinned

2.8 DYNAMIC KD-TREE

In this section, we discuss dynamic kd-trees that involve addition and deletion of points. This is a set of benchmarks in some sense for the management of dynamic trees. The dataset covered by these trees are subject to change over time, new points are added to the current set and some existing points are marked for deletion. The benchmark models a continuously changing domain, that needs to be updated real-time without additional memory copies or data transfers. We have modeled the benchmark based on iterative methods, the body of the loop is repeated in time. At certain fixed timesteps, we check for new points in the pending lists for addition and deletion. These lists are serviced, and the tree is updated if necessary. In our current implementation, tree adjustments are done lazily. But these are parameters that can be adjusted by the programmer to suit the needs of the application. Dynamic kd-trees are also summaries of partitions that are updated incrementally with new data. They can be used for applications with rapidly changing datasets, where frequent incremental load balancing is cheaper than full repartitioning for every addition and/or deletion.

The data structures used for maintaining dynamic datasets have different needs compared to linked lists for static data. They should support the following operations, with minimum overhead:

1. Lookup of points in the tree
2. Addition of points to the tree
3. Deletion of points from the tree
4. Split a leaf node when its weight exceeds bucket size
5. Remove leaf nodes that are empty
6. Combine leaf nodes with sum of weights < K*bucket size, $K < 1$

The data structure used for storing co-ordinates is different in this implementation to accomodate frequent addition and deletion of points. To reduce multiple deallocation, allocation and copying of points in an iteration, we use a dynamic data structure called point_blocks, which allows addition of new points and lazy deletion of unused points. The vector of indices is reallocated whenever additional memory is required. Memory for points is allocated in blocks, new blocks are added atomically when necessary. Points are deleted from point_blocks, by marking them as invalid. When all points in

a block are marked as invalid, it is removed from the set. The index vector stores the indices of valid points at any timestep. This set is updated as points are inserted or deleted. At any given timestep, the kd-tree, its partition and operations are defined on the current index vector, i.e. the valid/active points in the domain. Permutations, like Morton, Hilbert are defined on the active points. This type of partitioning where an initial partition is incrementally adjusted for load balance is used in many applications, both in scientific computing and databases. The programmer can request a full re-partition of point_blocks if necessary. We provide the pseudocode for such an application here 2.8 :

Algorithm 2.8 Dynamic_Pointset

```
1:  procedure Dynamic_tree_building(max_iter,step_size,num_samples)
2:      BUILD_DY_TREE()
3:      for doiter ← 1, max_iter
4:          if then!iter%step_size
5:              add_list ← NEW_POINTS(num_samples)
6:              del_list ← REM_POINTS(num_samples)
7:              bucket_list ← FIND_BUCKETS(add_list)
8:              UPDATE_INDEX_ARRAY(bucket_list, add_list)
9:              bucket_list ← FIND_BUCKETS(del_list)
10:             UPDATE_INDEX_ARRAY(bucket_list, del_list)
11:         end if
12:         if then!iter%2 ∗ step_size
13:             heavy_buckets ← LEAF_TRAVERSE()
14:             SPLIT_NODES(heavy_buckets)
15:             empty_buckets ← LEAF_TRAVERSE()
16:             DELETE_NODES(empty_buckets)
17:             merge_buckets ← LEAF_TRAVERSE()
18:             MERGE_NODES(merge_buckets)
19:         end if
20:     end for
21: end procedure
```

max_iter is the maximum number of iterations over which the loop executes. $step\_size$ is the timestep at which pending lists are checked. There are two pending lists - add_list and del_list. $num\_samples$ is the number of points added or deleted from pending lists every $step\_size$ iterations. The current tree is adjusted every $2 * step\_size$ iterations. The leaf nodes of the tree are buckets of granularity $>= 1$. $BUCKETSIZE$ is a parameter that can be tuned by the programmer. New points are added to the domain by searching for the smallest enclosing sub-cell in the tree. If the sub-cell is a leaf, then the point is added to a pending insertion list. If a matching leaf node was not

found, then starting from the smallest enclosing subcell, all nodes along the path to a bucket are expanded to include the point. Inclusion ensures node boundaries are not crossed, i.e. overlap is avoided. Inclusion by expanding nodes is done in parallel by sharing the list of new points (add_list) and dividing subtrees between threads. Once all points have found buckets, the new indices are inserted into the index array. Points are added to the dataset sequentially by starting from the last point_block. New blocks are added as point_blocks fill up. Pseudocode for addition of points is provided below in algorithms 2.9 and 2.10. The pseudocode shows the execution sequence for a single thread. Expansion of nodes is implemented by adjusting the corners of their bounding boxes.

Algorithm 2.9 Dynamic_Pointset

```
1:  procedure Addition_of_points(add_list)
2:      for do i ← 1, add_list.size
3:          n ← add_list(i)
4:          pos ← −1
5:          for do j ← 1, local_nodes.size
6:              cnode ← node_array[offset + j]
7:              if then CNODE.IS_INCLUDED(n)
8:                  if then cnode > pos
9:                      pos ← cnode
10:                 end if
11:             end if
12:         end for
13:         pos ← THREAD_REDUCE(pos, MAXIMUM)
14:         add_pos[i] ← pos
15:     end for
16: end procedure
```

Suppose there are T points in the add_list, N nodes in the tree, including leaf nodes and p threads. Let n be the number of points, d the number of dimensions and b the $BUCKETSIZE$. The average number of leaf nodes in the tree is $\lceil \frac{n}{b} \rceil$. The value of N is $2 * \lceil \frac{n}{b} \rceil$ for balanced kd-trees, which is the implementation in this thesis. These nodes are divided equally among threads. Each thread has at most $\lceil \frac{N}{p} \rceil$ local nodes. Threads traverse the entire list of pending points and searches for them in their set of local nodes. A point is included in a node if its co-ordinates are within the node's extents. For example, for $d = 3$, the condition to check for inclusion of a point with co-ordinates (x, y, z) is the following invariant for node n :

$$(x \geq n.min_x) \wedge (x \leq n.max_x) \wedge (y \geq n.min_y) \wedge (y \leq n.max_y) \wedge (z \geq n.min_z) \wedge (z \leq n.max_z) \tag{2.11}$$

For each point in the list, threads search for the smallest node (highest id) that contains it. If such a node exists, it is saved in a local variable. Threads synchronize to determine the smallest such node by computing the maximum over all node ids. This is the final node id that is saved for that point.

For T points, the cost of searching for the smallest containing node in the tree T_{aa} is:

$$T_{aa} = O(T * \frac{N}{p}) + T * O(p) \tag{2.12}$$

If the smallest containing node for a point is not a leaf, node extents need to be adjusted in the current subtree. Threads share the pending lists. Node adjustments are computed by threads for points lying within their subtrees, in parallel. The maximum number of node adjustments for any point is equal to the depth of the tree. The $ADJUST\_NODE$ function in the pseudocode is recursive and terminates when a leaf node is found. The depth of a tree with $\lceil \frac{n}{p} \rceil$ points and bucket size b is $log(\frac{n}{p*b})$.

Algorithm 2.10 Dynamic_Pointset

```
1:  procedure Find_buckets_points(add_list)
2:      for doi ← 1, add_list.size
3:          p ← add_list(offset + i)
4:          n ← add_pos(offset + i)
5:          if then!IS_LEAF(n)
6:              lpos ← ADJUST_NODE(n)
7:              add_pos(offset + i) ← lpos
8:          end if
9:      end for
10: end procedure
```

Deletion of points is also a two-step algorithm. The pending list called del_list is shared between threads. Suppose there are T points in the pending deletion list, N nodes in the tree, $BUCKETSIZE = b$ and p threads. Number of nodes owned by a thread is at most $\lceil \frac{N}{p} \rceil$. All threads iterate over the entire pending list, but search for points only within their local node sets. Threads search for the bucket containing a

Algorithm 2.11 Dynamic_Pointset

```
1: procedure Deletion_of_points(del_list)
2:     for do i ← 1, del_list.size
3:         n ← del_list(i)
4:         pos ← −1
5:         for do j ← 1, local_nodes.size
6:             cnode ← node_array[offset + j]
7:             if then CNODE.IS_INCLUDED(n)&IS_LEAF(cnode)
8:                 pos ← cnode
9:             end if
10:        end for
11:        pos ← THREAD_REDUCE(pos, MAXIMUM)
12:        del_pos[i] ← pos
13:    end for
14: end procedure
```

Algorithm 2.12 Dynamic_Pointset

```
1: procedure Del_bucket(del_list)
2:     for do i ← 1, del_list.size
3:         n ← del_list(i)
4:         b ← del_pos(i)
5:         pos ← −1
6:         for do j ← 1, local_nodes.size
7:             if then b == node_array[offset + j]
8:                 pos ← SEARCH_BUCKET(node_array[offset + j], n)
9:             end if
10:        end for
11:    end for
12: end procedure
```

point in its local node set. If such a bucket exists, its node id is saved. This is determined by a single synchronization step that computes the maximum of all such ids. In the second phase, buckets are searched to compute the exact location of a point within it. If the point exists in the bucket, it's index is deleted from the vector of indices.

Worst case computation cost for algorithm 2.10 T_{ab} is :

$$T_{ab} = O(T * log(\frac{n}{p * b})) \tag{2.13}$$

Total computation cost is T_a :

$$T_a = T_{aa} + T_{ab} \tag{2.14}$$

For T points, worst case computation cost of locating enclosing buckets in the tree is :

$$T_{ba} = O(T * \frac{N}{p}) + T * O(p) \tag{2.15}$$

For T points, the worst case cost of searching buckets is:

$$T_{bb} = O(T * b) \tag{2.16}$$

Total cost of deleting T points from a tree with N nodes and p threads is:

$$T_b = T_{ba} + T_{bb} \tag{2.17}$$

2.8.1 Tree Adjustment

One of the methods in which we construct new partitions is by incrementally modifying existing partitions. We define two operations on the kd-tree for this.

1. Split heavy buckets
2. Merge light buckets

At certain timesteps during the evolution of the pointset, the programmer has the option to adjust the existing kd-tree. This is done primarily for better load distribution between buckets. The frequency of invoking this routine depends on the application and the rate at which its workload evolves. Addition and deletion of points changes the number of points in leaf nodes and their offsets in the index array. Although these values are updated for leaf nodes, non-terminal nodes are untouched during addition/deletion. In our benchmark these nodes are updated lazily, depending on inputs from the application. For example, re-distribution can be done when the minimum bucket size drops to zero or the maximum bucket size is more than twice the average bucket size. Non-terminal nodes are updated bottom_up by computing their offsets and number of points recursively. The top nodes of the tree are assigned to threads which recursively update the weights of subtrees they own. Later the weights of nodes from top nodes to the root are updated by thread0. Each thread owns $\lceil \frac{n}{p} \rceil$ points in the domain, divided into buckets. The average depth of the subtree is $log(\frac{n}{p*b})$. For each node update, on an average $O(log(\frac{n}{p*b}))$ intermediate nodes need to be visited before arriving at leaf nodes. The direction of data propagation is from leaf nodes to the root. The most recent bucket sizes of leaf nodes are propagated up the subtrees.

The cost of updating non-terminals is:

$$T_{ad} = O(\frac{n}{p*b} * log(\frac{n}{p*b})) \tag{2.18}$$

where $log(\frac{n}{p*b})$ is the average depth of the tree for $BUCKETSIZE = b$.

Tree adjustments are performed once non-terminal data are consistent with leaf data. To adjust the tree, threads traverse the current node list in parallel and mark leaf nodes that are heavy (total weight > $BUCKETSIZE$). Heavy buckets are split recursively until leaves are within $BUCKETSIZE$. Non-terminal nodes that have total weight (sum of lower and upper leaf nodes) less than $BUCKETSIZE$ are marked for merging. Leaves are deleted by marking them *invalid* and replacing them with their parent nodes, which become new leaves. New leaves will have the same offsets as its their previous lower children and number of points that are sum of points in lower and upper cells. Both operations are simple to implement in parallel, without additional synchronization. When a node is split into two, the total number of points remains unchanged, which means it doesn't affect other nodes in the tree. Similarly, merging of leaf nodes, will not affect neighboring nodes in the tree. There is no need to update

non-terminals after these modifications to the tree, because we start from a consistent set and restrict adjustments to leaf nodes. Also, the total data size remains unchanged after adjustments. Things can get complicated if we allow adjustments that affect non-terminals, so we restrict ourselves to leaf nodes. The cost of tree adjustments is constant in the number of modified leaf nodes. Suppose K buckets are split or merged during an adjustment phase. Let T_{dd} be the cost of adjusting buckets :

$$T_{dd} = c * K \tag{2.19}$$

where c is the constant number of operations required for splitting a heavy bucket or merging light leaves.

2.8.2 Testcases

Testcases for evaluating the dynamic kd-tree partitioner are built around the pseudo-code provided in 2.8. The behavior of a dynamic dataset is modeled as snapshots of a point distribution that evolves over time. The test program runs iteratively, starting each iteration with a particular point distribution and two pending lists. After the lists are serviced, the underlying point distribution gets updated. The program starts with an initial tree building and partitioning phase.

The same datasets from the previous section were used to initialize the dynamic tree. New points are added to the domain by sampling from the bounding box at the root. Points for deletion are generated by sampling from the current dataset. Addition and deletion operations are performed on the most recent dataset at regular timesteps. For testcases in this section, new points are sampled every 100 iterations and appended to the lists. These are added to or deleted from the tree when the lists reach a certain size. Adjustments to the tree, i.e. splitting and merging are performed every 500 iterations. We have provided timing measurements for all operations which constitute one phase - tree building, addition of data, deletion of data and tree adjustments. There is also an option to rebuild the kd-tree from scratch if necessary. This can be used when the tree statistics, such as midpoint and median, that were used to build the initial tree, no longer hold. A full rebuild can be requested by the application at regular timesteps or based on performance data. We have a test case in the later sections for which we have

used these techniques. Although this test case comes from adaptive meshes in scientific computing, the behavior is found to be same. The dynamic tree-building program was executed for a maximum of 1000 iterations. All experiments were carried out on Stampede2 supercomputer. A single Intel KNL node was used for measuring performance of the multi-threaded program on shared memory. The results are tabulated below, for an initial dataset of size 1m points and 10m points for 3D and 10D.

#th	points	nodes	bsize	buildtime	addtime	deltime	adjtime	total
64	1m3D	90771	32	1.0326	0.25673	0.901217	3.9307441	78.8781
128	1m3D	90909	32	1.60241	1.39667	0.185382	0.714679	16.4273522
256	1m3D	90853	32	2.79706	3.32358	0.223829	1.74972	36.03697506
64	1m10D	94823	32	3.35145	1.13261	0.230857	0.850745	5.7140883
128	1m10D	94577	32	3.97817	1.32123	0.162733	0.71896	17.84933349
256	1m10D	94731	32	6.06864	2.76369	0.238967	1.15153	47.4207711
64	10m3D	289371	100	24.6541	15.1704	3.61651	16.9726	61.04216976
128	10m3D	289339	100	20.3154	17.6134	2.22621	15.591	87.5369396
256	10m3D	289737	100	23.7506	40.6131	2.2948	26.7047	164.1104614
64	10m10D	314629	100	52.9961	15.7457	4.19934	18.4795	91.9965334
128	10m10D	315361	100	58.2669	20.1613	2.85784	14.8437	129.6039031
256	10m10D	315277	100	73.2034	47.3685	2.81898	26.0353	226.457175

Table 2.11: Dynamic KD-tree construction time, midpoint splitter

There is no change in the tree building time. The results provided in table 2.11 are for uniform distribution with midpoint splitters. Addition of a point to the tree involves three main steps - identifying the leaf node which includes possible expansion of nodes along the path from the root of the largest subtree containing the point, linear search for location of the point within a bucket, deletion of current index vector and reallocation with newly included points. Deletion of an existing point is comparatively a cheaper operation. Tree adjustment depends on the amount of load imbalance - number of heavy buckets and light buckets. Since we are restricting modifications to the leaf nodes of the tree, this step has not been a serious bottleneck so far. The total time taken includes synchronization time and one parallel sorting step (for sorting the list of leaf nodes in the tree, arranged by their offsets). Poor quality of the kdtree can affect the total computation time, whatever that may be for the application. If datasets are derived from meshes, a poor-quality partition can increase the communication time during halo exchange. For pointsets without edges, the quality metric is not well-defined. If the kd-tree becomes unbalanced after some iterations, there may be long paths in the tree which affect the total search and update times. The definition of metrics is left to the programmer. If desired by the application, the partitioner will

rebuild the kd-tree using the current dataset and return a new order of leaf nodes. The graph in figure 2.14 is a graphical representation of the same results in table 2.11.

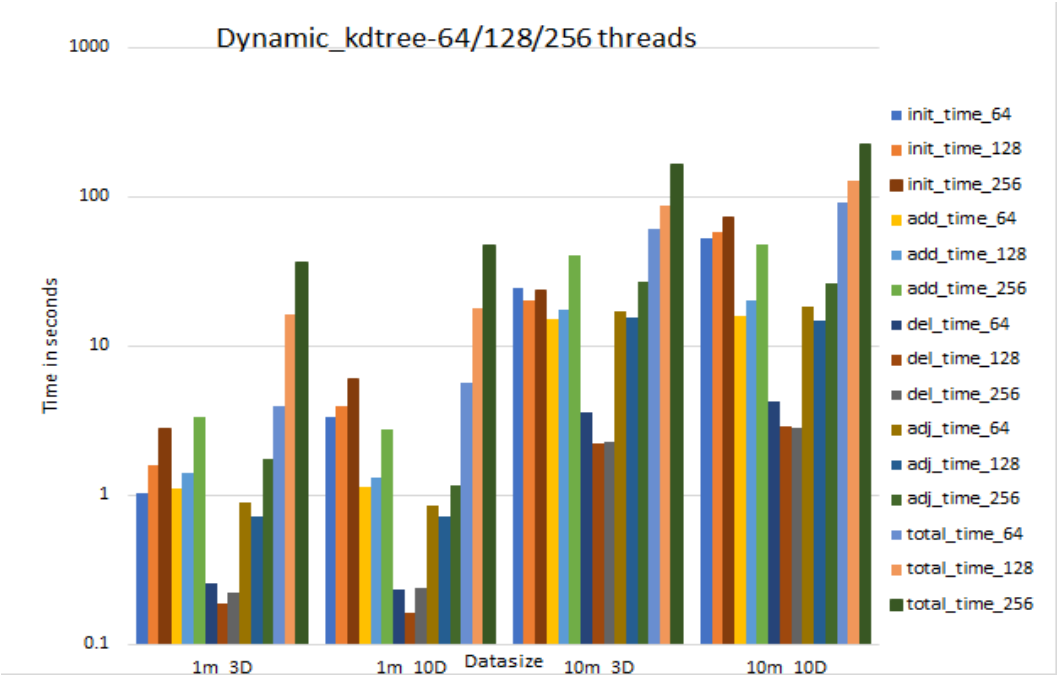


Figure 2.14: Dynamic KD-tree, uniform, scheduled and pinned

2.9 DISTRIBUTED KD-TREE

For large problems where it becomes difficult to assemble the entire mesh or dataset on a single node, we resort to distributed systems with multiple storage locations. This is also true when files require frequent backup to account for errors in software or failures in hardware. Distributed tree building handles scenarios where the data is stored in some random order across multiple nodes. The implementation is hybrid, where explicit messages are sent/received between nodes for synchronization and multi-threading is enabled on each node. The communication library used for off-node communication is MPI [85]. Multiple MPI processes with distinct ranks are instantiated and the kdtree tree is built by collaboration across ranks. We have used the words *process* and *rank* interchangeably in this document. The number of ranks per node depends on the node architecture. For KNLs we have used one MPI rank per node and at least one thread per CPU. For some multi-core nodes with larger caches and fewer CPUs, it may benefit if multiple ranks are placed on a node. Processes en-

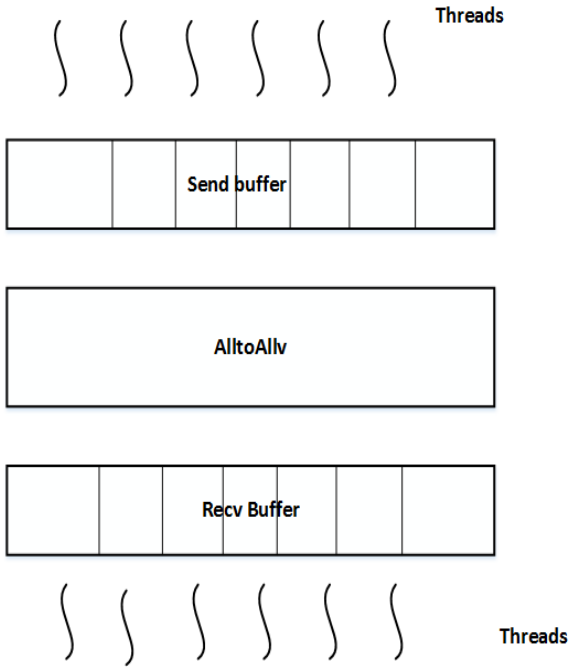


Figure 2.15: Parallel data packing and unpacking

sure memory isolation. Explicit messages need to be sent/received for on-node communication. The programmer can use his discretion for deciding the number of ranks per node. Since processes have different address spaces, explicit send/receive messages are required for computing splitters. This computation can become expensive if the algorithms requires them to synchronize over a network for every node creation. Moreover, additional buffers need to be allocated for sending and receiving messages between ranks. We try to reduce communication between processes by stopping tree building when there are enough tree nodes that can be distributed, without idling any process. Once nodes are assigned to processes, a data exchange routine is invoked. This first set of tree nodes are called *top-nodes*. The programmer has the option to reorder top nodes. In the default implementation, top nodes of the tree are built from a BFS traversal. We have applied Morton order permutation to these nodes. This can be replaced by other space-filling curve orders as well. Data is sorted partially by transferring points to processes that own enclosing top-nodes. In the worst case, there will be a full data-exchange at this stage. Data-exchange is done over several rounds to accommodate send/receive of large messages. We place an upper bound on the maximum number of bytes that can be transferred between any two processes in a single round. MAX_MSG_SIZE is a parameter that can be tuned by the programmer.

We have made packing and unpacking of MPI messages thread-parallel by creat-

ing non-overlapping regions in the buffers for different threads, shown in figure 2.15. Distinct offsets are used to isolate the buffer space owned by threads.

A single round of data-exchange performs the following operations :

1. parallel packing - off-node data are assigned to distinct threads. Threads compute the location of local data in the send buffer using a parallel-prefix operation. After determining offsets, off-node data are packed in parallel.

2. Collective data exchange - We have used a single MPI call for data exchange. Thread0 on every process gets involved in this AlltoAllv [86] communication. We tried to minimize the number of edges in the communication graph by aggregating messages. But if required, threads can start multiple simultaneous collective calls using different communication buffers.

3. parallel unpacking - Contents of the receive buffer are unpacked in parallel by threads, after determining offsets and counts.

After data exchange, tree building is resumed locally within each process. We have two versions here - static or dynamic subtrees. For the top nodes of the tree, the assumption is that data is randomly distributed across nodes. The points that lie within a tree node are usually scattered across many processes. Each MPI process stores the local dataset, arranged according to tree node indices. Splitters are decided collectively by all processes that participate in building the current tree node. We use collective operations such as $Allreduce$ and $Allgather$ for decisions on splitters. A pseudocode for a simple implementation is provided in algorithm 2.13, which computes midpoints for a set of top nodes. In our current implementation, top nodes are built by all processes in the group MPI_COMM_WORLD. In a general version of the algorithm, top nodes will be built by multiple MPI groups each working on non-overlapping subsets of tree-nodes. This program in algorithm 2.13 is executed by all MPI ranks in a group. All of them will have a private copy of the top-node list in the same order, but with local offsets and numbers of points. Some processes may not have any membership in a node, but we maintain empty nodes for now, because this portion of the tree should be globally consistent. A simple implementation of a distributed splitter is provided in algorithm 2.14. We have implemented, distributed midpoint and median selection algorithms.

Our implementation is hybrid, with pthreads used for computation within a process. For example, we use pthreads for computing local minimum and maximum for

Algorithm 2.13 Initialize_Dist_tree

```
1: procedure init_dist_tree(pointset)
2:     init_list ← root
3:     while
4:     init_list.size() < THRESHOLD
5:     cnode ← init_list.pop()
6:     SPLIT_DIST(cnode)
7:     l ← LOWER(cnode)
8:     u ← UPPER(cnode)
9:     !IS_LEAF(l)
10:    init_list.push(l)
11:    !IS_LEAF(u)
12:    init_list.push(u)
13: end procedure
```

the pseudocode given here. A similar implementation is used for median selection splitter. Let P be the number of processes in the group, ranked from $0, ..., P-1$, p the number of threads per rank, numbered $0, ..., p-1$. Let N be the total number of points in the domain where each rank has at most $\lceil \frac{N}{P} \rceil$ elements in some random order. In algorithm 2.14 *compute_min* and *compute_max* are functions that are local to each process. These functions are multi-threaded and compute the local bounding box widths. The cost T_l of computing local box widths for one node in d dimensions is :

$$T_l = O(d * \frac{N}{P * p}) + O(d * p) \tag{2.20}$$

The $O(d * p)$ communication cost is for combining local minimum and maximum values in d dimensions.

After computing the corners of local sub-regions, all ranks share the co-ordinates of their bounding boxes through two $allreduce$ [86] invocations. Assuming a PRAM delay model for interprocess communication, the cost of an $allreduce$ collective operation is $O(\alpha * logP + \beta * c * logP)$, where α is the latency per message, β is the transfer cost per byte and c is the number of bytes transferred per process. Two such $allreduce$ operations with short messages is the communication overhead per node-split when points are distributed. In the actual implementation, we have aggregated data for a set of top-nodes to reduce the number of collective operations.

Distributed median computation is more expensive compared to midpoint splitters. Each rank samples a set of points, a larger set of samples is created by concatenating

Algorithm 2.14 Dist_Splitter

```
1: procedure split_dist(n)
2:     for do i ← 1, NDIM
3:         local_min[i] ← COMPUTE_MIN(i)
4:         local_max[i] ← COMPUTE_MAX(i)
5:     end for
6:     global_min ← MPI_AllReduce(MINIMUM, local_min)
7:     global_max ← MPI_Allreduce(MAXIMUM, local_max)
8:     d ← COMPUTE_MAX_DIM(global_min, global_max)
9:     m ← global_min[d] + (global_max[d] − global_min[d])/2
10: end procedure
```

all samples, using an $allgather$ invocation. The communication overhead of $allgather$ is $O(\alpha * logP + \beta * c * PlogP)$, where c is the message size per rank. The final message size is $c * P$, which is the concatenation of messages from all processes.

The load balancing routine is a replicated computation at all ranks. Provided all ranks perform the same computation, the final load assignment is deterministic. It is a simple greedy assignment of load to bins. A similar pseudocode is provided in section 5.3.4 in Chapter4. We used $allreduce$ and $parallel\_prefix$ to determine some statistics regarding the load distribution, such as $maximum\_load$, $total\_load$, $minimum\_load$ and $rank$ of a node in the top-node list.

The data-exchange following load balancing is the most communication intensive part of the implementation since it may require a full data exchange in the worst case. If the data exchange takes place over r rounds, the cost of a single round is the time taken to transfer the largest message between any two nodes. Let T_{comm_i} be the maximum communication cost in round i, the total communication cost T_{comm} is,

$$T_{comm} = \sum_{i=1}^{r} T_{comm_i} \tag{2.21}$$

Data exchange ensures that a process is home to all points that lie within the nodes it owns. It is left to the programmer to decide whether to pick a static subtree per process or have dynamic subtrees that are modified iteratively to accommodate adaptive partitions. Final permutation of buckets is distributed across processes in sorted order. The keys/ids of points on $P_i < P_j$ where $i < j$.

2.9.1 Testcases

The first testcase was used to analyze the performance of a distributed static kd-tree implementation with one MPI process per KNL node and $>= 64$ threads. These experiments are strong scaling, with the same dataset, but with increasing number of nodes and CPUs. The number of MPI ranks was varied from $16 - 256$. There are three values for number of threads - 64, 128 and 256. The total number of cores ranges from $[1024 - 16384]$. Total number of threads ranges from $[1024 - 65536]$. We used a uniform distribution with 1billion 3D points sampled from $[1, 1000000000]$ to test this implementation. The STL random distributions were used to generate uniform samples within a certain range. Time taken for building the distributed kd-tree is divided into three components:

#ranks	threads	top_nodes	lb_time	transfer_time	local_subtree	total
16	64	9.63115	0.000856329	3.986062	61.9971158	75.61518413
32	64	3.7515925	0.000914826	2.400112	11.6435904	17.79620973
64	64	3.2753625	0.001162399	1.436562	6.000608	10.7136949
100	64	1.983095	0.001199494	0.9691276	2.813082	5.766504094
128	64	2.18574	0.001254128	3.789026	2.7854376	8.761457728
150	64	1.750285	0.00121716	6.310382	2.320145	10.38202916
200	64	1.62908	0.001136252	5.757782	1.8497324	9.237730652
256	64	1.696865	0.001835986	6.03519	1.8342938	9.568184786
16	128	6.174555	0.001240974	6.510466	48.8319268	61.51818877
32	128	4.1130975	0.001121844	3.866482	9.5244262	17.50512754
64	128	3.103545	0.001386724	2.328766	3.1429028	8.576600524
100	128	2.7942025	0.001428744	2.227704	2.7292164	7.752551644
128	128	2.7067625	0.001594928	3.804966	2.145137	8.658460428
150	128	2.6590975	0.001387716	11.03982	1.8678144	15.56811962
200	128	2.5737725	0.001774634	5.33363	1.5153696	9.424546734
256	128	2.594755	0.001698394	6.977644	1.36721	10.94130739
16	256	4.3497675	0.00260731	12.75062	38.2026036	55.30559841
32	256	3.2551125	0.002927878	7.05198	12.1406462	22.45066658
64	256	2.7115925	0.003721518	4.08766	5.6249762	12.42795022
100	256	2.54733	0.003735332	3.444074	3.8978448	9.892984132
128	256	2.5200075	0.003117306	5.604658	2.7971492	10.92493201
150	256	2.4508075	0.003417202	11.419	2.9936656	16.8668903
200	256	2.4658225	0.0033248	7.607246	2.9016206	12.9780139
256	256	2.5727175	0.003032592	6.224126	2.286735	11.08661109

Table 2.12: Distributed KD-tree construction time, midpoint splitter

1. Top nodes: This is the distributed code section which requires data-sharing across MPI ranks. The number of top nodes is a configurable parameter. For the test-

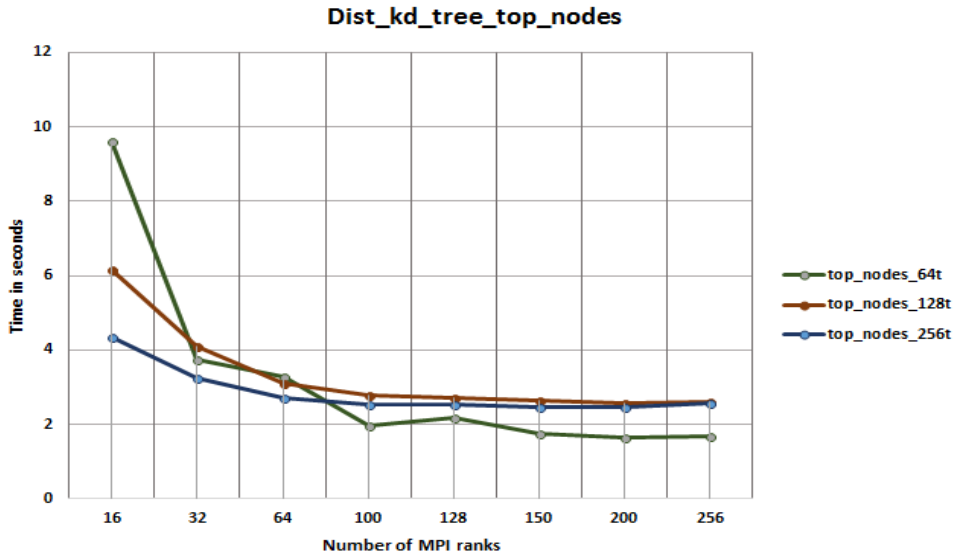


Figure 2.16: Distributed KD-tree top-nodes construction

case in this section, this value was 256. The graph in figure 2.16 is the cost of building top nodes for increasing number of MPI ranks and threads. Maximum $BUCKET\_SIZE = 100$.

2. Load balancing and data exchange: The load balancing routine assigns top nodes to MPI ranks. After mapping nodes to ranks, non-local data are transferred to owning processes. This is a relatively expensive stage. In the worst case, the communication pattern is an all-to-all, where every process has at least one message to send/receive from every other remote process. Communication is performed in stages to accomodate exchange of large datasets. Maximum message size in each stage is capped by the parameter MAX_MSG_SIZE. For the testcases here, $MAX\_MSG\_SIZE = 10000$ bytes per thread.

3. Local subtrees: We re-use the routines from the static parallel kdtree implementation discussed in the previous sections.

The graph in figure 2.17 is the sum of all components, including load balancing and data exchange. The values on the y-axis of the graph are based on log scale. The graph shows some variation in scaling after 100 MPI processes. The pre-dominant cost in this region is data exchange compared to local subtree building. The time taken for data transfer depends on various factors, such as maximum number of messages, maximum message size, latency and bandwidth.

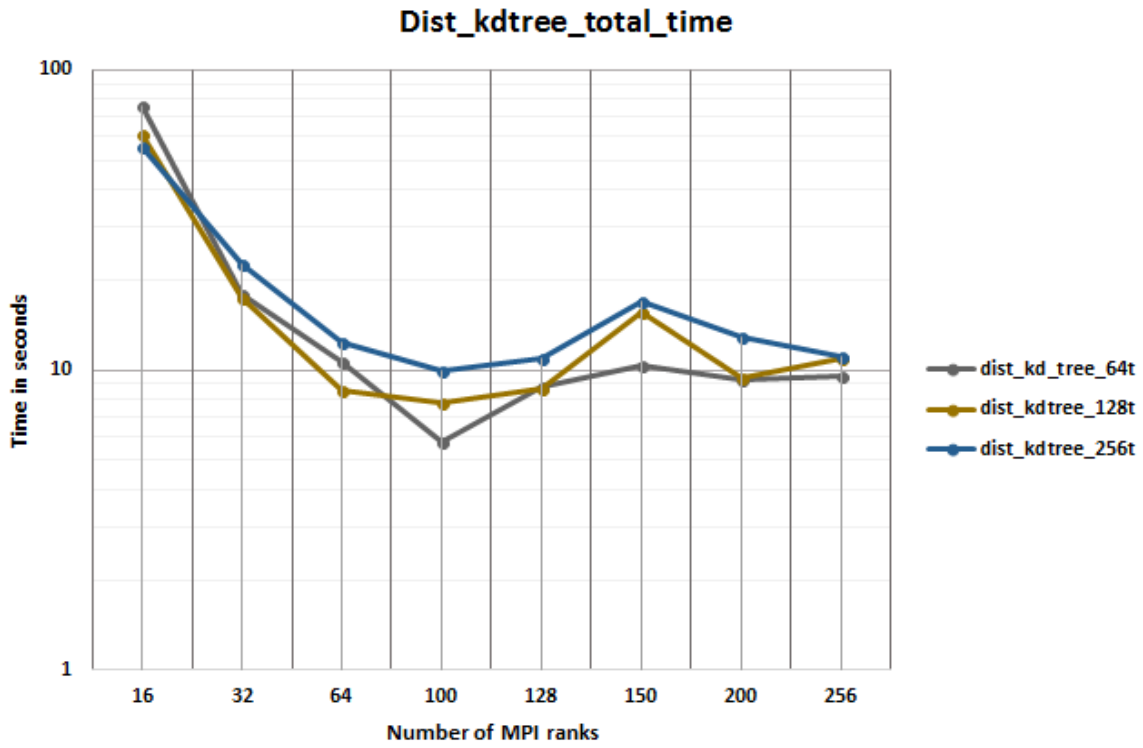


Figure 2.17: Distributed KD-tree total time

2.10 PARALLEL QUICKSORT

This section has a brief description of the parallel quicksort implementation we used as a component in the partitioner. We used this implementation for sorting samples and co-ordinates in the splitter routines and re-order leaf nodes in the dynamic kd-tree implementation. Pthreads are used for partitioning and exchanging data. Unsorted data is assumed to lie in a contiguous vector. We have not used any synchronization primitives to control access to the shared vector. Threads are assigned non-overlapping sub-regions in the vector. Execution is controlled using thread barriers and offsets are computed using parallel prefix. The recursion in quicksort is a binary tree, with each node representing an unsorted section of the array. The problem of parallelizing quicksort is equivalent to scheduling the nodes of this binary tree on threads, satisfying their node dependencies. The dependencies are simple, the children of a node can be scheduled to run only after the parent node has completed execution. The children of a node are created when a node is partitioned around a pivot. All elements less than and equal to the pivot constitute the left child, the remaining elements form the right child.

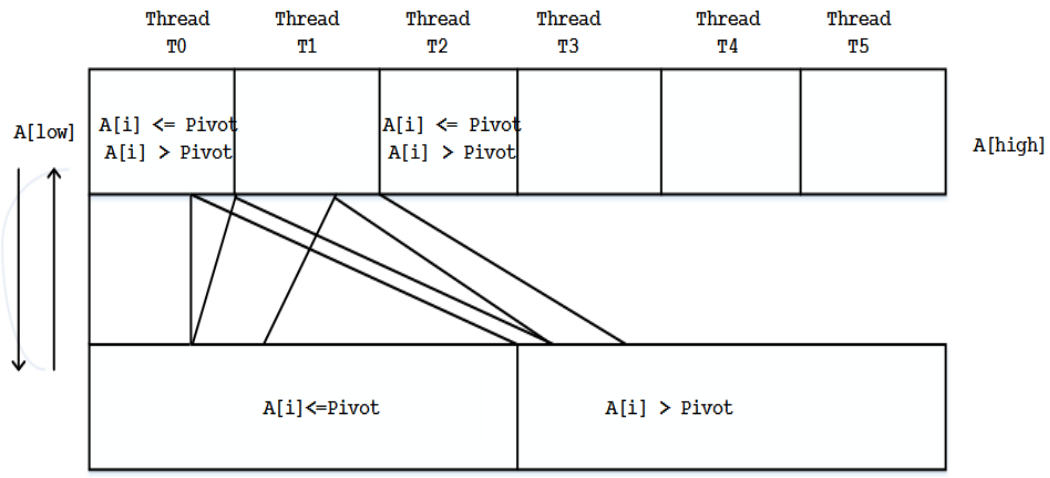


Figure 2.18: Data decomposition for parallel quicksort,top nodes

Data partitioning for sorting follows the same model as tree-building, both problems being recursive. The top nodes of the recursion tree operate on large sections of unsorted data. The time-consuming step of parallel quicksort is rearranging the array into two sections - less than and equal to pivot and greater than pivot. For the top nodes, data partitioning is done collaboratively by threads until there are enough independent nodes that can be executed by different threads. One thread picks a pivot and shares it with other threads. The unsorted section is divided between threads in a load balanced manner. For a section of size n elements and p threads, each thread works on a piece of size at most $\lceil \frac{n}{p} \rceil$. Threads rearrange portions of the array they own. These partially sorted chunks are copied to a temporary buffer, concatenated and relocated in the source vector. Figure 2.18 shows pivoting for top nodes using a temporary buffer. We use reduction and parallel prefix to compute the source and destination indices. The top recursion ends when there are enough sub lists to be distributed between threads, at least one sub list per thread. These sub lists have non-overlapping offsets, and they are sorted in parallel. The total sorting time depends on the number of elements and maximum depth of the recursion tree.

Choosing good pivots that split the array into roughly equal parts is an important criterion for reducing the depth of recursion. The selection of pivots is randomized for this reason. There are similar implementations of parallel quicksort optimized for GPUs. Suppose the top recursion stops when there are $r * p, r >= 1$ non-overlapping subsets. Let T_{qs} be the average cost of parallel quicksort. The average cost of building the top nodes T_{qp} is:

#threads	num_points	init_time	build_time
64	100000	0.211413	0.0268961
128	100000	0.3876418	0.02666706
256	100000	0.9851252	0.02966264
64	1000000	0.253258	0.136962
128	1000000	0.486924	0.2733168
256	1000000	1.020914	0.2215962
64	10000000	0.3304084	2.227984
128	10000000	0.5166522	2.708028
256	10000000	1.024476	1.783118
64	100000000	0.9890994	21.646294
128	100000000	0.8240288	21.078394
256	100000000	0.9782016	39.97772
64	250000000	2.141636	91.25146
128	250000000	1.576726	56.5622
256	250000000	1.537538	64.83902
64	500000000	4.077758	97.29324
128	500000000	2.87157	117.51094
256	500000000	2.375654	203.4084
64	1000000000	6.570702	270.5132
128	1000000000	5.533122	213.0476
256	1000000000	4.39892	357.143

Table 2.13: Sorting time for different data sizes and thread counts

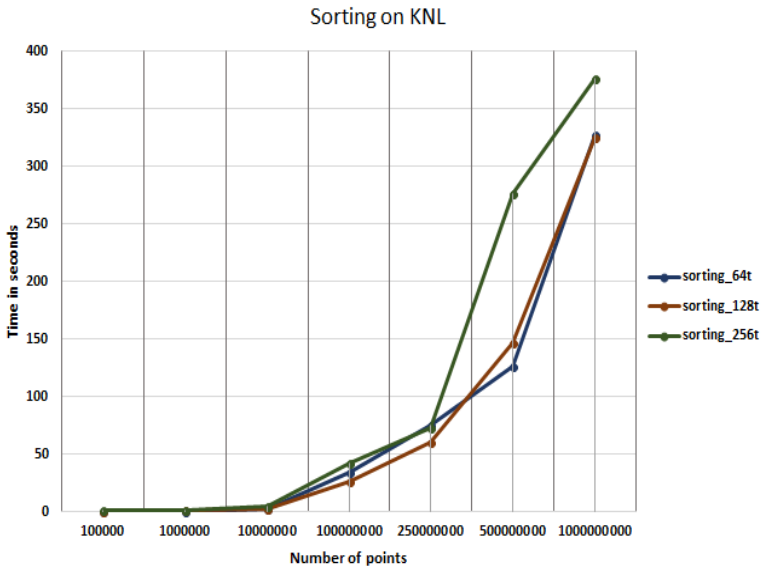


Figure 2.19: Sorting time for different data sizes and thread counts

$$T_{qp} = O(\frac{n}{p} * log(r * p)) \tag{2.22}$$

where $O(\frac{n}{p})$ is the cost of rearranging data at level i in the recursion tree.

Assuming the top nodes are load balanced, where each thread has $\lceil \frac{n}{p} \rceil$ elements on average for $r = 1$, the cost of sorting at the lower levels of the recursion tree is :

$$T_{lp} = O(\frac{n}{p} * log(\frac{n}{p})) \tag{2.23}$$

$$T_{qs} = T_{qp} + T_{lp} + O(p)) \tag{2.24}$$

where $O(p)$ is any additional synchronization.

All experiments in this section were carried out on Intel KNL nodes [80]. The results presented in this section are for sorting doubles. Inputs range from 100000 to 1000000000 doubles, all generated from uniform random distributions. Threads can migrate between cores during the top level of recursion since unsorted sections can be anywhere along the array. Thread scheduling is left to the operating system. Once threads are assigned lists in local queues, there are pinned to cores. Each thread works on restricted sections of the array, pinning them to cores increases cache re-use after this point. For large data sizes and multiple threads per core, the performance degrades, due to threads sharing lower level cache. There is possible thrashing for these configurations. Tests were conducted using three thread counts - $64, 128, 256$ and 64 CPUs. Inputs are random doubles generated from a uniform distribution. Some of them are more expensive than others to sort. We have reported the average of 5 distributions. The observations are tabulated in table 2.13. A graph with the results is shown in figure 2.19. The performance of sorting drops for large datasets, especially for 256 threads. This is probably due to thrashing in L2 which is shared by four threads. KNL was used in cache-quadrant mode where the entire MCDRAM is used as cache.

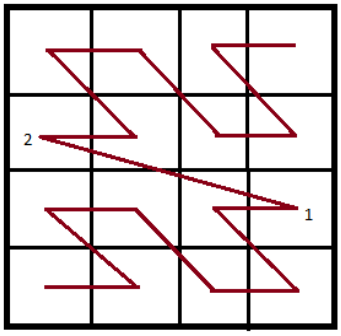


Figure 2.20: Morton Order

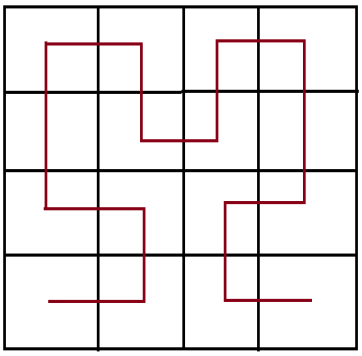


Figure 2.21: Hilbert Order

2.11 SPACE-FILLING CURVES

After the kd-tree is constructed, its nodes are traversed using space-filling curves. In the next two chapters we discuss these traversals in detail for 2D and 3D points. Define a discrete space-filling curve as the inverse of a function (F) defined on points along a curve C to points in d-dimensional space R^d. For every point in R^d, there is a corresponding point on the curve that identifies it. These functions can be defined algebraically, as geometric series, with different geometric ratios for each curve. There are three properties that define any function that is a space-filling curve:

1. Space-filling: The curve should be space-filling, i.e it should exist on every point in the domain.
2. Single visits: Each point in the domain should be visited exactly once. No edges along the traversal should be visited more than once. SFC traversal of a domain is a solution to the travelling salesman problem (TSP) [87]. The solution using SFCs is a Hamiltonian on the points, in which no vertex is visited more than once [87].
3. Surjective: The function that maps from R^d to curve C is not one-to-one. There are points in the domain R^d for which membership to sub-domains is ambiguous. For example, in the figure 2.21, if a point exists at the center of the domain it can belong to any of the four sub-domains, according to the way the curve is defined. But once a decision is taken, the point should not be revisited.

Geometric constructions of these curves are recursive. A pattern is defined at the top-level, which is scaled down and repeated in the sub-domains, subject to some other transformations (like rotation and reflection). The curves differ based on the number of sub-domains at each level of recursion and the repeating pattern. They are

well-defined for 2D and extended to higher dimensions algebraically, but the recursive geometric constructions are not clear for dimensions greater than 2. An increase in the number of dimensions affects the degrees of freedom in the curve and its possible transformations. There are several algebraic definitions of space-filling curves, some of which are more complicated to construct compared to others. Sagan [13] has a good description of SFCs and fractals. The most commonly used SFCs are Hilbert, Peano and Morton. Hilbert and Morton divide the domain into 2^d sub-domains at each level and apply the repeating pattern connecting them at edges. Peano curve has 3^d sub-domains at each level.

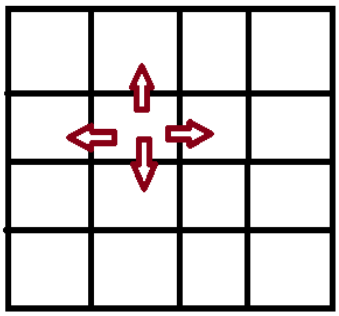


Figure 2.22: Communication pattern example - 5pt stencil

Several metrics are defined to compare the quality of different curves. But, since we use these curves to partition data for parallel applications, metrics are defined accordingly. For a given communication pattern, we define *locality* of a space-filling curve as the number of communicating neighbors that lie outside a given volume, usually a hypercube. This value is related to the edge-cut (communication volume) of a partition. An example for nearest neighbor exchange in 2D is shown in the figure 2.23. Data is exchanged between grid elements that share a common face. If a space-filling curve has few outgoing edges for a given volume of points, it is considered to have better locality. In order to have good locality, it is necessary for the curve to maintain the following condition: adjacent points on the curve are neighbors in higher dimensions. The definition is not clear for points in 3D. A 3D-curve is *face-continuous* if any two adjacent points share a common face. A curve can also be edge-continuous where adjacent points share a common edge. For our 3D SFC, we consider both as neighbors in 3D. Any other pair of points is treated as a discontinuity in the curve. If a curve is used to traverse a set of points without any communication pattern defined on them, it is difficult to define *locality*. One of the metrics that can be used is the average length of the curve that traverses a given volume in R^d. Consider three points p_1, p_2 and p_3,

where d_{ij} is the Euclidean distance between any pair of points (p_i, p_j). If the curve at p_1 picks p_2 as the next point instead of p_3, although $d_{13} < d_{12}$, it can be considered as a discontinuity. Between any two curves c_1 and c_2, the better curve should be able to traverse the same volume with fewer discontinuities. The total length of the curve that covers all points in volume V should be lower for c_1 compared to c_2 for it to be a better-quality order. The length of a space-filling curve is the sum of Euclidean distances of all adjacent points on the curve.

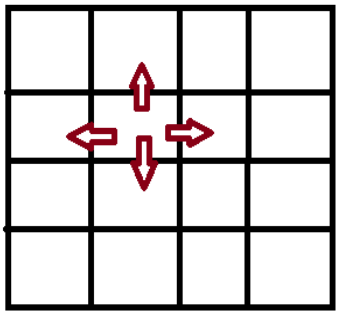


Figure 2.23: Communication pattern example - 5pt stencil

Several metrics are defined to compare the quality of different curves. But, since we use these curves to partition data for parallel applications, metrics are defined accordingly. For a given communication pattern, we define *locality* of a space-filling curve as the number of communicating neighbors that lie outside a given volume, usually a hypercube. This value is related to the edge-cut (communication volume) of a partition. An example for nearest neighbor exchange in 2D is shown in the figure 2.23. Data is exchanged between grid elements that share a common face. If a space-filling curve has few outgoing edges for a given volume of points, it is considered to have better locality. In order to have good locality, it is necessary for the curve to maintain the following condition: adjacent points on the curve are neighbors in higher dimensions. The definition is not clear for points in 3D. A 3D-curve is *face-continuous* if any two adjacent points share a common face. A curve can also be edge-continuous where adjacent points share a common edge. For our 3D SFC, we consider both as neighbors in 3D. Any other pair of points is treated as a discontinuity in the curve. If a curve is used to traverse a set of points without any communication pattern defined on them, it is difficult to define *locality*. One of the metrics that can be used is the average length of the curve that traverses a given volume in R^d. Consider three points p_1, p_2 and p_3, where d_{ij} is the Euclidean distance between any pair of points (p_i, p_j). If the curve at p_1 picks p_2 as the next point instead of p_3, although $d_{13} < d_{12}$, it can be considered as

a discontinuity. Between any two curves c_1 and c_2, the better curve should be able to traverse the same volume with fewer discontinuities. The total length of the curve that covers all points in volume V should be lower for c_1 compared to c_2 for it to be a better-quality order. The length of a space-filling curve is the sum of Euclidean distances of all adjacent points on the curve.

Algorithm 2.15 Slicing Algorithm

```
1:  procedure SLICING
2:      avg_wt ← avg(w1, w2, ..., wn)
3:      max_wt ← max(w1, w2, ..., wn)
4:      cur_wt ← 0
5:      cur_bin ← 0
6:      for do i ← 1, n
7:          if then cur_wt + wi < (cur_bin + 1) * avg_wt + max_wt
8:              cur_wt ← cur_wt + wi
9:          else
10:             cur_bin ← cur_bin + 1
11:         end if
12:     end for
13: end procedure
```

The space-filling curves in figures 2.20 and 2.21 are Morton and Hilbert curves that cover the same set of points in 2D. The domain is symmetric and has grid dimensions that are powers of 2. Both curves can be sliced into segments of almost equal lengths, where the length of a segment is the sum of the weights of all points on the curve. If each segment is assigned to a partition, the maximum load imbalance of an assignment is the weight of the heaviest mesh element. As far as load balance is concerned, there is no difference between segments generated by different space-filling curves. However, the edge-cuts or communication volume are different, especially for partitions of lengths that are non-powers of 2. There is a direct relationship between discontinuities and edge-cut of partitions. Since a discontinuity places non-local points next to each other on a curve, when it is sliced and assigned to processes, these add to the total number of out-going edges from the partition. For any set of non-neighbors in a partition, the total contribution to the edge-cut is the sum of their neighbors. This can be reduced if neighbors are assigned to the same partition by a curve that is continuous. The discontinuities in Morton order are shown in the diagram 2.20. Points 1 and 2 are not neighbors but they may be assigned the same partition which will increase communication volume. The algorithm we used for slicing the curve into roughly equal length segments is given in algorithm 2.15. The slicing method is a greedy assignment

of points to bins, where bins are partitions. Each bin has a desired weight, equal to the average weight of the distribution. If the assignment is optimal, maximum bin weight will be equal to the desired value. The membership of a point in a bin is decided based on its current weight, weight of the point and the desired weight. The maximum weight of any bin depends on the weight distribution, which in-turn depends on the order of traversal. Morton and Hilbert orders produce different orders of weights for the same dataset. But the maximum bin weight for greedy assignment is at most $2 * M$, where M is the maximum weight of any point [87].

CHAPTER 3: 2D SPACE-FILLING CURVES

3.1 2D TRAVERSAL RULES

In this section, we discuss a general 2D Space-filling curve that can be applied to irregular point distributions and resembles a Hilbert curve when the domains are square symmetric [13]. We compare this space-filling curve to Morton order. At every internal node of the kd-tree, its resolution is checked. A node is considered *fully-resolved* in 2D, if it has four grandchildren or sub-domains that are generated from hyperplanes perpendicular to each other. For the fully resolved cases, traversal rules are optimized to match Hilbert rules. Rules are recursively defined and derived by applying transformations to a base rule. If a node is *partially-resolved*, which can happen for domain sizes that are uneven and not powers of two, we pick traversal rules that reduce the distance between adjacent points on the curve. The rules are explained in detail later. This curve can also be used for traversing domains with shapes that are not squares. In our implementation, the following transformations can be applied to the base rule. The inputs to a transformation function are a set of subcells and a rule specifying a traversal order on them. The output is the permuted set of subcells:

1. Reflection: This transformation reverses the order of traversing subcells. Geometrically, it corresponds to the mirror reflection of a rule. A rule may be reflected along any of the two axes. For example, reflection can change clockwise traversal to anti-clockwise.

2. Rotation: This transformation traverses sub cells after applying a circular shift to the base order, that corresponds to a 90 or 180-degree rotation about any axis.

Transformations are composite functions. A series of transformations can be applied to the base rule, to generate all possible traversal orders. We picked a base rule with X as major axis and Y as minor axis. It is possible to pick another base rule and arrive at a different set of rules. But it can be proved without difficulty that both sets are equivalent. There exists some set of transformations that convert one set of rules to the other and vice versa. In short, reflection and rotation are minimal transformations required to generate Hilbert like space-filling curves in 2D. The sizes of sub cells decrease at every level of recursion.

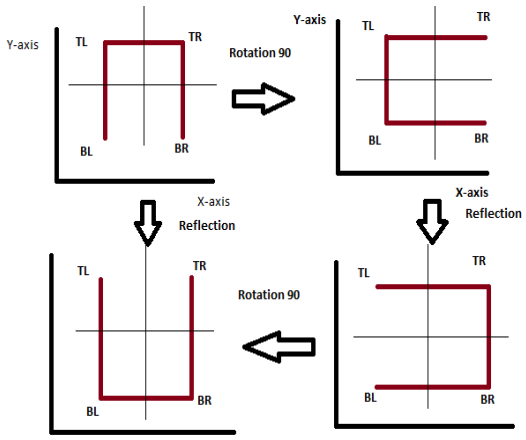


Figure 3.1: Transformations of 2D Hilbert curve

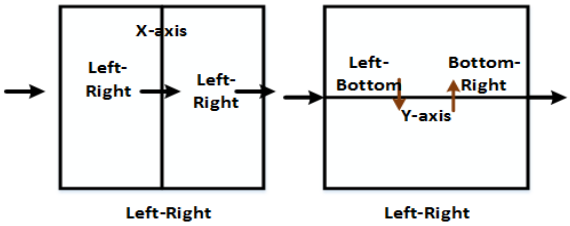


Figure 3.2: 2D Refinement rules

The diagrams for a base rule and some of its transformations for fully resolved nodes are provided in the figure 3.1:

The rules for partially refined nodes are provided in figure 3.2. For symmetric grids with dimensions that are powers of 2, all internal nodes will pick Hilbert rules, because all of them will be fully-resolved. For odd dimensions and irregular distributions, there will be bounding boxes in the domain that are partially resolved. For such cases, traversals for sub cells are assigned using the rules in 3.2. There are two such rules - when the entry direction is perpendicular to the splitting dimension or parallel to it. If the entry direction is parallel to the splitting dimension, there are two options for traversing the sub cell. We pick the option which is closest to entry. If it is perpendicular, there are no ambiguities in how the sub-domains are refined.

3.2 EMPIRICAL MEASUREMENTS

Programs to construct the general SFC and its evaluation were written in C/C++. There are routines to estimate quality metrics (load balance and communication volume) for a partition and communication pattern. The communication pattern and weights (optional) have to be provided as secondary inputs to the program. We have used VTK [88] for visualizing the SFC. The first set of testcases are empirical measurements used to compare the quality of SFC partitions to those generated by multilevel implementations in Metis and in Scotch. All the testcases and experiments in this sec-

tion use sequential kd-tree and space-filling curve implementations.

Here we provide a brief explanation of tuning parameters in Metis:

1. ufactor - This is used to specify the maximum load imbalance that can be tolerated in terms of number of vertices (mesh cells). An integer value of 30 (default) generates partitions with maximum load imbalance of 1.03. We used two values for this parameter, the least possible value 1, and 30.

2. niter - This parameter decides the number of iterations in the refinement phase. The values we used are 10 (default) and 400.

Scotch computes a mapping of the input graph onto the machine. Since our performance metrics do not include machine topology, we used a complete graph. Scotch was configured to use a multi-level scheme for each bipartition. The default options in Scotch seemed exhaustive enough. Besides, in the default case, Scotch generates two independent partitions and outputs the best of the two. The only tuning parameter we used was the load balance constraint which was set to 0.01, for a maximum of 1% imbalance. Although recent versions of Scotch can compute a direct k-way partition instead of a bipartition, we found its load balance and edge-cut values much worse than the bipartition implementation for our inputs.

The choice of refinement algorithms we used in Metis and Scotch are described below:

1. Greedy refinement (Metis) : For any coarse graph Gi , this algorithm greedily picks a random boundary vertex Vj that results in maximum decrease in edge-cut without violating load balance constraints. This vertex is moved to the neighboring partition.

2. Scotch : The default refinement strategy in Scotch uses a combination of several schemes. All refinements are computed on a band graph of width 3, which is a graph consisting of only those vertices at a maximum distance of 3 from the separators. The refinement phase does 40 passes of diffusion followed by several iterations of Fidducia-Matteyses until convergence.

3.2.1 Structured Grids

3.2.1.1 Uniformly Refined

#procs	maxload	max_degree	max_comm_vol
256	3456	13	1140
512	1728	13	804
1024	864	14	564
1500	590	13	464
2048	432	14	396
3000	295	14	320
4096	216	14	276
6000	148	14	226
8192	108	14	192

Table 3.1: Degree and Edge cuts for Morton Partitions of Structured 768X1152 grid

#procs	maxload	max_degree	max_comm_vol
256	3456	9	1092
512	1728	9	710
1024	864	9	540
1500	590	9	464
2048	432	9	350
3000	295	10	324
4096	216	9	264
6000	148	9	222
8192	108	9	170

Table 3.2: Degree Edge cuts for Hilbert Partitions of structured 768X1152 grid

#procs	maxload	max_degree	max_comm_vol
256	3456	8	716
512	1728	9	572
1024	864	8	356
1500	590	9	464
2048	432	9	284
3000	295	10	326
4096	216	8	176
6000	148	9	226
8192	108	9	140

Table 3.3: Degree Edge cuts for gensfc Partitions of structured 768X1152 grid

A 2D rectangular grid with 1152 co-ordinate values in the y-dimension and 768 values in the x-dimension was used as testcase. We chose arbitrary values for $k1$ and $k2$

here, to evaluate the effectiveness of GenSFC in partitioning asymmetric structured grids. A midpoint splitter was used along the longest dimension. The communication pattern is a simple 9-pt stencil and we assume that a cell sends one byte of different data to each of its neighbors. Therefore, in this case, the communication volume of a partition is exactly equal to its edge-cut. Tables 3.1, 3.2 and 3.3 compares the quality of partitions produced by GenSFC against those generated by Morton and Hilbert curves. We do better than both Morton and Hilbert when the number of processors is a power of two. We have better partitions for most cases since we split along the dimension of maximum spread unlike Hilbert and Morton which alternate between splitting dimensions. When the number of processors is not a power of two, a GenSFC partition is spread across multiple subtrees leading to an increase in its surface area. In these cases, the communication volume is comparable to Hilbert partitions.

#procs	maxload	max_degree	max_comm_vol
128	5164	8	684
180	3679	8	588
256	2592	8	504
380	1767	9	434
512	1308	8	350
750	900	9	308
1024	659	11	298
1500	450	9	204
2048	329	9	188
3000	225	9	144
4096	164	9	134

Table 3.4: Degree Edgecuts for Metis partitions of structured 768X1152 grid

The partitioning time for space-filling curve partitions was found to be lower than that of Metis and Scotch. Moreover, for a given input mesh it is constant. For multilevel methods, partitioning time increases with increasing number of partitions.

The next set of experiments were designed to better understand the behavior of multilevel schemes implemented in Metis and Scotch and the impact of tuning parameters on resulting partitions. Since SFCs guarantee partitions with load imbalance of at most 1 element, we tried tightening the load balance constraint in Metis for a fair comparison. We set $ufactor = 1$ to decrease the maximum load imbalance to 1.001%. Metis partitions are much worse with higher values for communication volume and degree. The behavior of the algorithm seems to be unpredictable, results are shown in table 3.4. The partitions improved when we increased the number of iterations of refinement, but

the performance was not consistent, especially for large partition counts. One possible way to get around this erratic behavior would be to generate multiple sets of partitions and choose the best one like Scotch does. We did not explore that option. The best Metis partitions we obtained for this grid, are tabulated in table 3.5 alongside the best Scotch partitions. These results were obtained for default values for all its input parameters. Scotch partitions seem to have better load balance than Metis, but the maximum communication volume and maximum degree are worse. Both multilevel schemes do better than Morton and Hilbert curves for all cases. The general SFC algorithm does much better than Metis and Scotch when the number of partitions is a power of 2. For non powers of 2, the two paradigms are comparable, with better load balance provided by the general SFC algorithm, and much faster partitioning time.

#procs	maxload	max_degree	max_comm_vol
128	5160	8	760
180	3677	9	656
256	2583	8	506
380	1738	10	462
512	1292	8	388
750	881	10	330
1024	646	10	290
1500	441	11	258
2048	323	10	210
3000	221	11	166
4096	162	10	152

Table 3.5: Degree Edgecuts for Scotch Partitions of structured 768X1152 grid

3.2.1.2 Unstructured Meshes

Traditional SFCs fail to generate good quality partitions of unstructured meshes due to their arbitrary point distributions. GenSFC traverses random point distributions quite well. Figure 3.3 is GenSFC on a set of randomly located points in 2D. One of the meshes that has posed a challenge to geometric partitioners is that covering a volume (a shell). We applied the general SFC algorithm to such a mesh commonly used in climate simulations. The meshes we used come from the Model for Prediction Across Scales (MPAS) project [89]. They are constructed using Voronoi tessellations on the surface of a sphere and the cells are mostly hexagons with some pentagons and heptagons. A cell is represented by a point at its center of mass. MPAS currently uses the multilevel methods of Metis to partition these meshes. The partitions are generated off-line

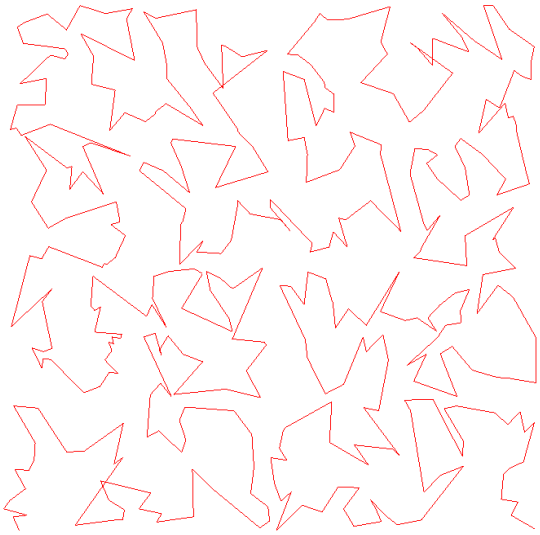


Figure 3.3: GenSFC on irregular points

and read from a file at the start of the simulation. Since the cells lie on the surface of a sphere (earth), we cannot use a 2D SFC directly on the cell centers. We first projected the points onto a plane. To obtain good quality partitions, we chose a projection with minimum distortion. It is impossible to project the entire sphere onto a plane without any distortion. So we divided the sphere into two hemispheres (north and south), projected them separately and generated SFCs for each half. We assigned initial directions to the kd-trees so that the SFCs are connected. The northern hemisphere was traversed from $Left\_to\_Right$ and the southern hemisphere from $Right\_to\_Left$. We used the Azimuthal Equidistant projection [90] for each hemisphere. This projection preserves area and, hence, preserves locality reasonably well. If ϕ and λ indicate the latitude and longitude values of a cell center, the equations for projection are

$$x = \rho * cos(\theta) \tag{3.1}$$

$$y = \rho * sin(\theta) \tag{3.2}$$

For the northern hemisphere,

$$\rho = (\frac{\pi}{2} - \phi) * R \tag{3.3}$$

$$\theta = \lambda \tag{3.4}$$

For the southern hemisphere,

$$\rho = (\frac{\pi}{2} + \phi) * R \tag{3.5}$$

$$\theta = -\lambda \tag{3.6}$$

3.2.2 Uniformly Refined

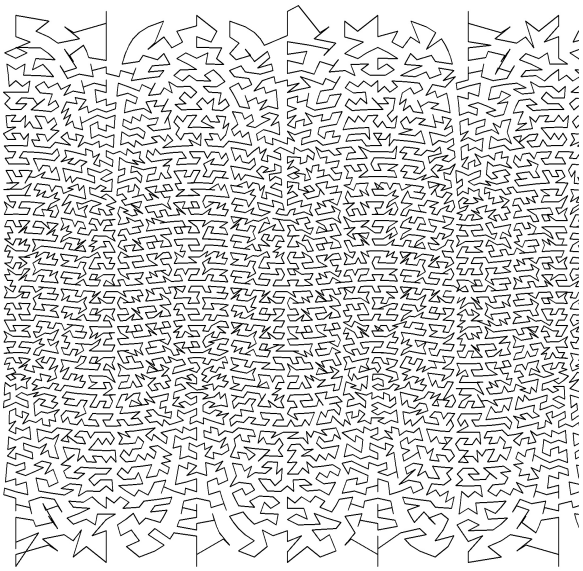


Figure 3.4: GenSFC on MPAS 30KM atmospheric mesh

Now the inputs to the general SFC algorithm are projected x and y co-ordinates of cell centers. Tables 3.6, 3.7 and 3.8 compare the quality of partitions generated by three different SFCs : Morton, Hilbert and general SFC. Unlike structured grids, the general SFC does much better than Morton and Hilbert curves even when the number of partitions are not powers of 2. These partitions were generated by splitting along the midpoint of the dimension of maximum spread. We generated Metis and Scotch partitions for comparison with our SFC partitions. For Metis, we have only included the best values we obtained for $ufactor = 30$ and $niter = 10$. The comparison between Metis and Scotch partitions is tabulated in tables 3.9 and 3.10. Metis seems to produce partitions with consistently better edge-cut and degree values and Scotch partitions have better load balance. The general SFC does better than Metis and Scotch for most

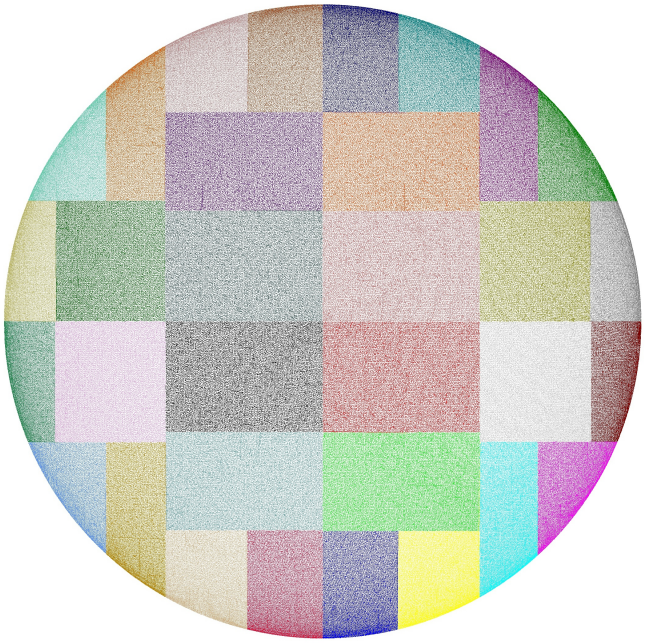


Figure 3.5: Uniformly Refined Atmospheric Mesh Partitioned using GenSFC

of these cases, especially at large values of #procs. The best Metis partitions were obtained at the cost of a small load imbalance and they are comparable to the general SFC partitions when the number of processors is a power of 2. For non powers of 2, our partitions are slightly worse than the best Metis ones.

#procs	maxload	max_degree	max_comm_vol
128	5121	11	2104
180	3641	12	1680
256	2561	13	1481
380	1725	13	1180
512	1281	13	1076
750	874	12	880
1024	641	12	752
1500	437	13	632
2048	321	13	548
3000	219	13	456
4096	161	13	384

Table 3.6: Morton Order on Uniformly Refined 30KM Mesh

#procs	maxload	max_degree	max_comm_vol
128	5121	8	1180
180	3641	8	1038
256	2561	8	734
380	1725	8	762
512	1281	8	612
750	874	9	554
1024	641	9	456
1500	437	10	378
2048	321	9	290
3000	219	13	292
4096	161	9	230

Table 3.7: Hilbert Curve on Uniformly Refined 30KM Mesh

#procs	maxload	max_degree	max_comm_vol
128	5121	7	717
180	3641	10	889
256	2561	8	485
380	1725	9	615
512	1281	8	370
750	874	10	448
1024	641	9	268
1500	437	10	324
2048	321	9	196
3000	219	11	228
4096	161	9	131

Table 3.8: GenSFC on Uniformly Refined 30KM Mesh

#procs	maxload	max_degree	max_comm_vol
128	5164	8	684
180	3679	8	588
256	2592	8	504
380	1767	9	434
512	1308	8	350
750	900	9	308
1024	659	11	298
1500	450	9	204
2048	329	9	188
3000	225	9	144
4096	164	9	134

Table 3.9: Metis partitions on uniformly refined 30KM Mesh

#procs	maxload	max_degree	max_comm_vol
128	5160	8	760
180	3677	9	656
256	2583	8	506
380	1738	10	462
512	1292	8	388
750	881	10	330
1024	646	10	290
1500	441	11	258
2048	323	10	210
3000	221	11	166
4096	162	10	152

Table 3.10: Scotch partitions on uniformly refined 30KM Mesh

#nodes	load/task	max_deg_inter	max_comm_vol_inter
24	878	76	2713
32	659	52	1486
40	527	77	2137
64	329	53	1084
100	210	85	1436
200	105	87	1018

Table 3.11: Clustering(32 tasks per node) of Metis Partitions for 30KM unstructured Mesh

3.2.3 Two-level Partitions

In the final set, we explored the the quality of a two-level layout (nodes and cores) using partitions generated by Metis, Scotch and our SFC algorithm. The SFC algorithm used a midpoint splitter along the dimension of maximum spread. The coarse partitions were created by grouping fine partitions in the order of their partition numbers (index). The number of partitions in a group corresponds to the number of cores per node and the number of groups is equal to the number of nodes. We measured the degree and edge-cut values for inter-node communication. As expected, Metis partitions fail in this regard because Metis has no notion of groups. The quality of clustering for Metis worsens with increasing number of partitions per node. We have analytical results for 32 Metis partitions per node, tabulated in table 3.11. We have assumed that inter-node communication is fully serialized. Therefore, the degree and edge-cut values for the results in this section are computed as the sum of outgoing messages and edges over all tasks on a node. All values in table 3.11 are the maximum observed

numbers across all nodes. We utilized the mapping option in Scotch to estimate the quality of clustering in our SFC algorithm. We used a two- level tree topology as input to Scotch. The inter-node edges were made ten times as heavy as the intra-node edges to reflect the hierarchical structure of the machine. Scotch results are tabulated in 3.12. As shown in 3.13 our SFC partitions match up to the mapping produced by Scotch for most node counts. The degree and edge-cut values of the SFC partitions are slightly worse than the Scotch mapping when the number of partitions is not a power of 2. We are able to generate good quality two-level partitions naturally, without any additional work.

#nodes	load/task	max_deg_inter	max_comm_vol_inter
24	862	51	1611
32	646	51	1410
40	517	58	1396
64	323	53	976
100	207	58	882
128	162	58	714
200	104	62	638
256	81	59	552

Table 3.12: Clustering(32 tasks per node) of Scotch Partitions for 30KM unstructured Mesh

#nodes	load/task	max_deg_inter	max_comm_vol_inter
24	854	58	2150
32	641	53	1248
40	513	63	1590
64	321	53	955
100	205	68	1098
128	162	60	717
200	103	72	862
256	81	58	489

Table 3.13: Clustering(32 tasks per node) of GenSFC partitions for 30KM unstructured Mesh

3.2.4 Adaptively Refined Meshes

The general SFC algorithm can handle adaptive meshes quite well. Figure 3.6 shows the SFC on the northern hemisphere of an adaptive atmosphere mesh with 163842 cells. The points are XY projections of cell centers as explained in the previous section. Cell

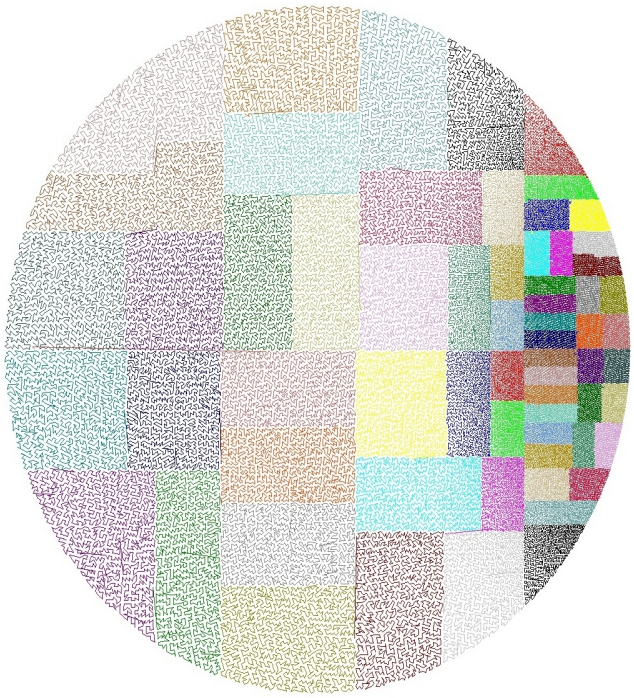


Figure 3.6: GenSFC partitions of unstructured adaptive mesh

shapes are mostly hexagonal and the refinement region extends over approximately 60 degrees of latitude/longitude. Our results are consistent with previous sections. The general SFC does better than Morton and Hilbert in all cases. In this testcase, we used an exact median splitter along the dimension of maximum spread. Like in the case of uniformly refined unstructured meshes, general SFC partitions seem to match up to Metis partitions. Edge-cuts for SFC partitions are higher than those for the best Metis partitions when the number of partitions is not a power of 2, but the SFC load balance guarantees are better. Tables 3.14, 3.15 and 3.16 compares Metis and Scotch partitions for the same mesh. Scotch partitions have better load balance than Metis, but edge-cuts are worse. The general SFC displays good clustering for the adaptive mesh as well. The maximum total inter-node degree and edge-cut values for Metis, Scotch and GenSFC partitions were computed. Tables 3.17, 3.18 and 3.19 show the computed values for groups of 32 tasks per node. The clustering property of general SFC is comparable to the mapping produced by Scotch for adaptive unstructured mesh, especially when the number of nodes is a power of two.

#procs	maxload	max_degree	max_comm_vol
128	1281	13	1024
256	641	12	720
380	432	12	620
512	321	13	508
750	219	12	440
1024	161	12	380
1500	110	14	308

Table 3.14: Morton Partitions of unstructured adaptive atmospheric mesh

#procs	maxload	max_degree	max_comm_vol
128	1281	8	584
256	641	8	472
380	432	9	384
512	321	8	334
750	219	9	214
1024	161	9	202
1500	110	10	170

Table 3.15: Hilbert Partitions of unstructured adaptive atmospheric mesh

#procs	maxload	max_degree	max_comm_vol
128	1281	9	390
256	641	9	280
380	432	10	305
512	321	10	218
750	219	9	216
1024	161	10	160
1500	110	10	154

Table 3.16: GenSFC Partitions of unstructured adaptive atmospheric mesh

#nodes	load/task	max_deg_inter	max_comm_vol_inter
10	526	75	2056
16	329	50	984
20	263	81	1605
32	164	52	768
50	105	84	1033
64	82	54	548

Table 3.17: Clustering(32 tasks per node) of Metis for adaptive unstructured mesh

#nodes	load/task	max_deg_inter	max_comm_vol_inter
10	517	52	1153
16	323	51	938
20	258	54	859
32	162	56	724
50	103	57	615
64	81	54	502

Table 3.18: Clustering(32 tasks per node) of Scotch for adaptive unstructured mesh

#nodes	load/task	max_deg_inter	max_comm_vol_inter
10	513	55	1500
16	323	53	942
20	258	59	1180
32	162	55	710
50	103	65	812
64	81	57	515

Table 3.19: Clustering(32 tasks per node) of GenSFC for adaptive unstructured mesh

3.2.5 Running Time of Simulations

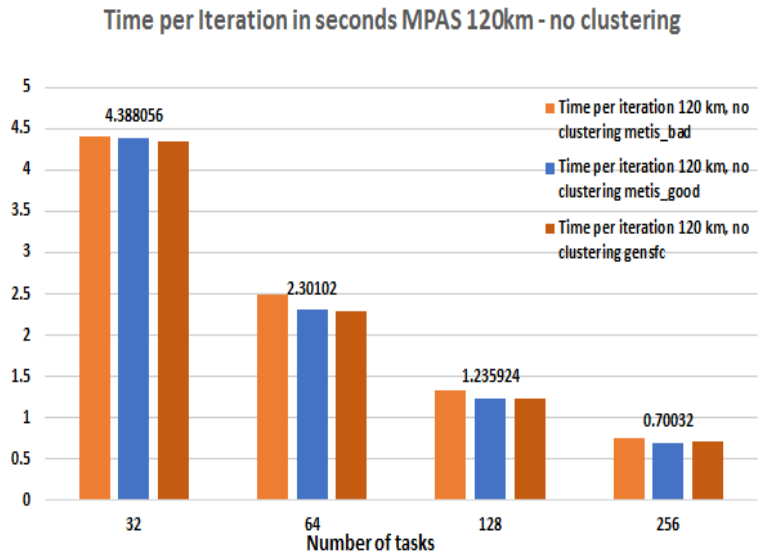


Figure 3.7: Execution Times for Unstructured MPAS mesh with resolution 120km

We validated our calculations by executing the MPAS- Atmosphere core (with physics turned off). MPAS uses Metis by default. The partitioner was modified to accept GenSFC partitions. The execution time reported is the time per iteration of the dynamics module, averaged over 15 iterations. All experiments were carried out on Mira, a Blue Gene Q (BG/Q) machine [91] at Argonne National Laboratory [92]. The MPAS

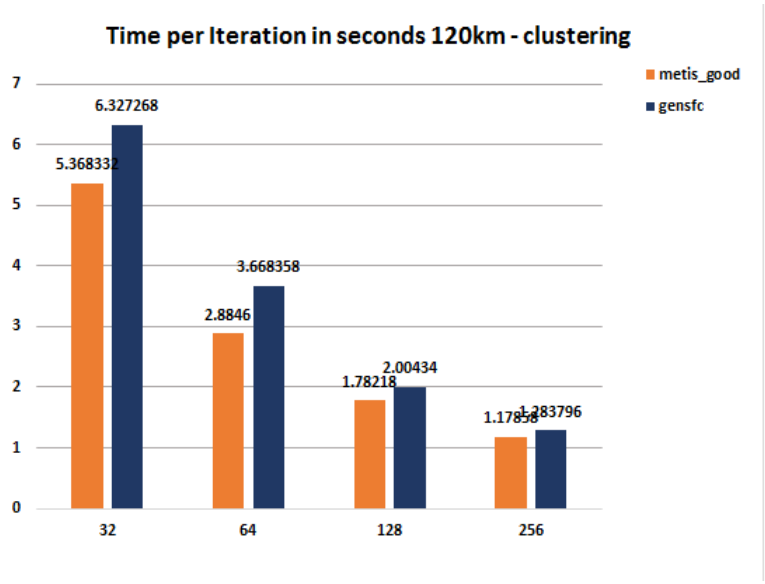


Figure 3.8: Execution Times for Unstructured MPAS mesh with resolution 120km

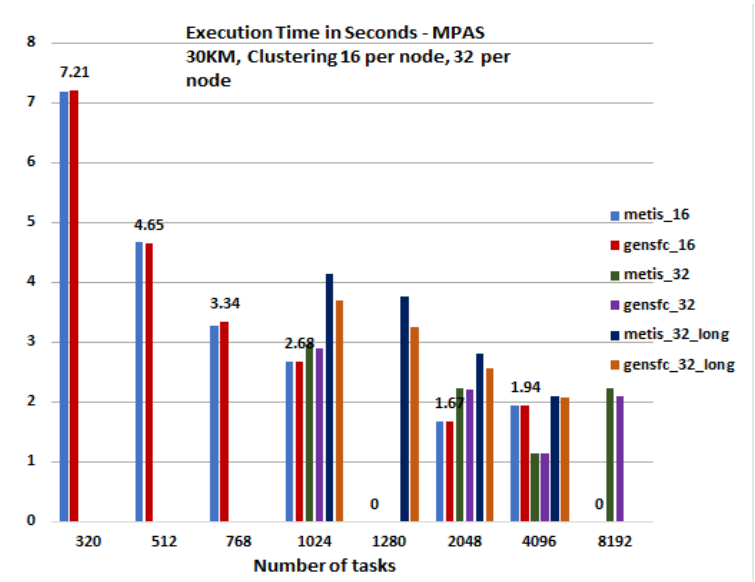


Figure 3.9: Execution Times for Unstructured MPAS mesh with resolution 30km

code is distributed with one MPI rank per process. The version we used was not multi-threaded. We found the intra-node communication cost on Mira to be as significant as off-node communication for this assignment. Both values were comparable for short messages. The intra-node message latency for 0 byte messages was found to be 3.01 microseconds, while the inter-node message latency was 3.40 microseconds. Therefore, for short messages we did not notice a big difference between the measured execution times. The contributing factor to the communication time in this case is the maximum degree of any partition. The maximum degree distributions for the different partitioners are similar. But, the intra-node bandwidth is much higher than the inter-node bandwidth. To improve the overall performance of the application, we increased the cut-off for MPI eager protocol for intra-node communication using the BG/Q environment variable $PAMID\_EAGER\_LOCAL$. The default value for this parameter on BG/Q is 4096 bytes, same as that for inter-node communication. When this cut-off was increased to 1MB ,all MPI intra-node messages are were sent eagerly. The plot in figure 3.7 is the measured execution time for three partitioners - a bad Metis partition ($ufactor = 1$), a good Metis partition and GenSFC partitions. Figure 3.8 shows the execution times with 16 MPI ranks per node after changing $PAMID\_EAGER\_LOCAL$ value. A good metis partition was used for comparison. Although total maximum degree and maximum edge-cuts were better for GenSFC partitions, the performance was worse than the best Metis partitions for few nodes. This is probably due to the increased problem size and therefore memory accesses per node. We ran similar experiments for the high resolution 30 km mesh. The graphs in figure 3.9 show the measured execution times per step, for Metis (good) and SFC partitions. There are two configurations for MPAS in this experiments - short messages and long messages. In this experiment we increased the message size by increasing the number of halo layers. For two halo layers, messages are short, therefore, we don't see any impact of clustering here. To demonstrate the difference between partitions, we increased the number of halo layers to eight. The lower execution time per step achieved by clustering in SFC partitions is seen at large messages.

CHAPTER 4: 3D SPACE-FILLING CURVES

4.1 3D TRAVERSAL RULES

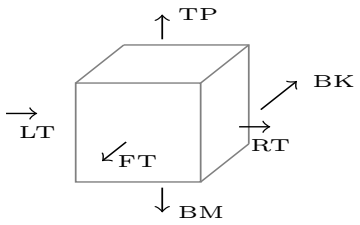


Figure 4.1: 3D bounding box with axes and corners

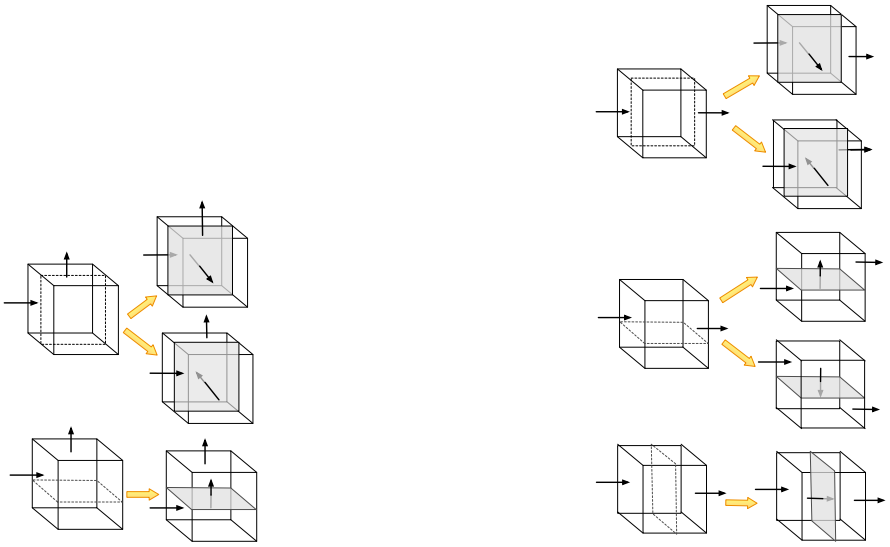
This section describes rules for a 3D space-filling curve, along the lines of the 2D curve. Non-terminal nodes of the kd-tree are axis-parallel bounding boxes, figure 4.1.

A bounding box in 3D has 6 faces and 8 corners. Faces and corners are labeled according to the orientation of the axes. For example, top-left corner of the box in figure 4.1 is labeled FTL where F stands for $front$ (position along the Y-axis), T for top, (Z-axis) and L for $left$,(X-axis). At the intermediate nodes, boxes are split into two, non-overlapping sub cells of smaller volumes.

After tree-building, the curve is constructed by traversing it top-down, closely following a set of 3D rules. Rules are uniquely identified by two directions (entry, exit) and two corners. Traversal is initialized by specifying a rule for the root node. Traversals for sub-domains are generated top-down, independently, connected by their entry and exit faces, edges or corners. In this section, we describe traversal rules optimized for points in 3D.

Traversals are defined with respect to a bounding box of arbitrary size. Although there are 30 different ways to enter and exit a box with 6 faces, we consider two base cases; the remaining cases are generated as permutations of the two traversals. The base cases are called *cis* and *trans* depending on whether entry and exit faces are adjacent to or opposite to each other. The base cases are explained briefly in 4.2a and 4.2b.

1. Cis - This case covers traversals where entry and exit faces are adjacent to each



(a) Cis Base case

(b) Trans Base case

Figure 4.2

other. The splitting plane can be perpendicular to any of the three dimensions. Entry and exit directions for sub cells are generated by connecting them along their common face (splitting plane). For example, suppose for cell A, a traversal is specified as $entry = left$, $exit = top$ and $entry\_point = FBL$ and let the splitting hyperplane be XY. The hyperplane along XY creates two sub cells, one containing points whose z co-ordinates are $<=$ to the splitter (lower sub cell) and the other containing points that are strictly greater than the splitter (upper sub cell). The lower sub cell is assigned directions $entry = left$, $exit = top$ and the upper sub cell gets $entry = bottom$ and $exit = top$ for the next level of recursion. This is shown in the bottom figure of 4.2a. The top figure of 4.2a has the same $entry$, $exit$ and $entry\_pt$ values, but the splitter is the XZ plane. Here sub cells can be assigned directions $entry = left$, $exit = back$ and $entry = front$, $exit = top$ respectively. The other option is to assign directions $entry = left$,$exit = front$ and $entry = back$,$exit = top$. Both are shown in 4.2a.

2. Trans - This case covers traversals where $entry$ and $exit$ faces are opposite to each other. If the splitting plane is perpendicular to the entry direction, traversals for sub cells are generated without any ambiguity. For example, suppose a cell is assigned directions $entry = left$ and $exit = right$, and the splitting plane is YZ perpendicular to $left - right$ (X) axis, the lower sub cell is assigned directions $entry = left$, $exit = right$ and the upper sub cell directions $entry = left$,$exit = right$. This is the case covered in figure 4.2b.

We allow additional degrees of freedom by specifying *orientation* for the curves. For the same $entry$, $exit$ directions and $corners$, there are more than one ways to orient the curve from entry to exit. The figure in fig 4.3 shows two ways of traversing a bounding box from FBL corner to FBR corner, using Hilbert curves. All variations of Hilbert curves can be generated from a base-rule and a minimal set of transformations. A base-rule for a set of 8 sub cells and one of its variations is shown in 4.3. Similarly, for a set of four sub cells we use the 2D rules explained earlier. Remaining rules are generated from the base-rule by applying two transformation functions - rotation and reflection. Consistent with the discussion in previous sections, transformation functions are composite. Once can find equivalence between any two 3D Hilbert curves using these set of transformations. In the figure 4.3, the base-rule for 8 points is the traversal on the left. The traversal on the right is obtained by a 90^0 rotation of the base-rule about the $X-axis$, followed by a reflection about the $Z-axis$ or $Y-axis$, depending the choice of rotation (+/-). We have restricted ourselves to curves that are symmetric, i.e. if one face of a cell is traversed in the clockwise direction, the other is traversed in anti-clockwise direction and vice versa. If this restriction is removed, there can be many more curves with the same $entry$, $exit$ directions and $corners$, some of them non-contiguous (not face or edge contiguous). When curves are symmetric there is more similarity in the traversals which reduces average length of the curve covering a volume.

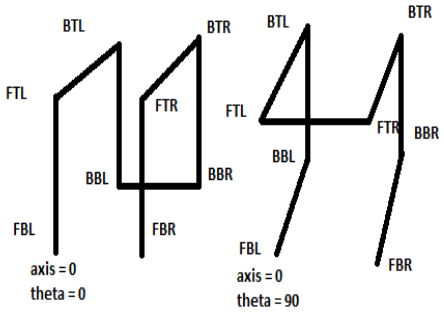


Figure 4.3: Two possible orientations of a Hilbert curve from $left$ to $right$

If the set of points form a $2^k \times 2^k \times 2^k$ regular grid then our rules generate a Hilbert-like curve in 3D that is *face-continuous*. The curve is smooth and adjacent boxes share a common face. For non-powers of two, adjacent boxes may be connected by a common face, a common edge or in the worst case, a common vertex.

(a) 3D Hilbert Rule (b) 2D Hilbert Rule

Figure 4.4: 2D and 3D Hilbert Rules

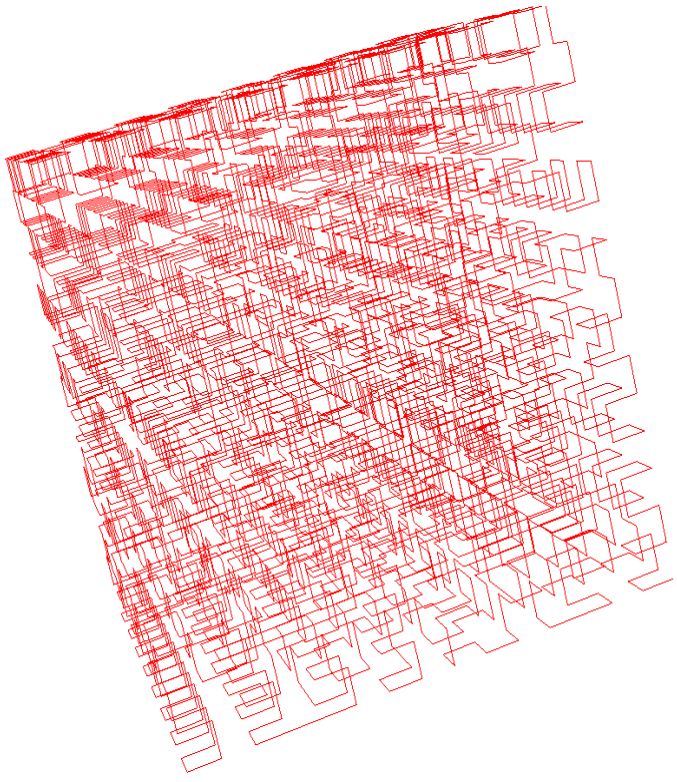


Figure 4.5: A 3D Space-filling Curve

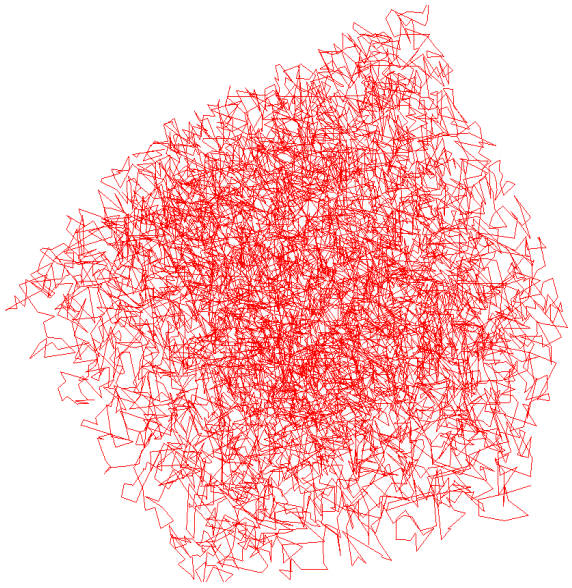


Figure 4.6: A 3D Space-filling Curve on Irregular Distribution

Algorithm 4.1 Optimizations

```
1:  procedure LOOKAHEAD OPTIMIZATIONS(node *n)
2:      r ← n.resolution
3:      if then n == 8
4:          HILBERT3DRULE(n)
5:      else
6:          if then n == 4
7:              HILBERT2DRULE(n)
8:          end if
9:      end if
10: end procedure
```

We have included optimizations in this implementation that identifies groups of 8 or 4 sub cells of approximately equal aspect ratio. In such cases, since the number of sub cells are powers of 2, the curve benefits from applying a symmetric Hilbert-like rule to the subset. Such groups are identified by checking the resolution of a node during traversal. A cluster of 8 sub cells has 3 splitting hyperplanes that are perpendicular to each other. A cluster of 4 sub cells has two perpendicular hyperplanes. These optimizations are briefly explained below :

4.2.1 Two-lookahead Optimization

This optimization is applied if a cell n is split into 8 almost equal sub cells. A shortcut link is added to the tree, from n to its eight grandchildren. A permutation of the 3D Hilbert rule, shown in figure 4.4a is used to traverse this set.

4.2.2 One-lookahead Optimization

Suppose a group of 4 sub cells is identified at cell n which lie on a plane, we use one of the 2D Hilbert rules, shown in figure 4.4b, to traverse this set. The base rule is rotated appropriately, using permutations, depending on the entry and exit directions at n.

4.3 PUTTING IT TOGETHER

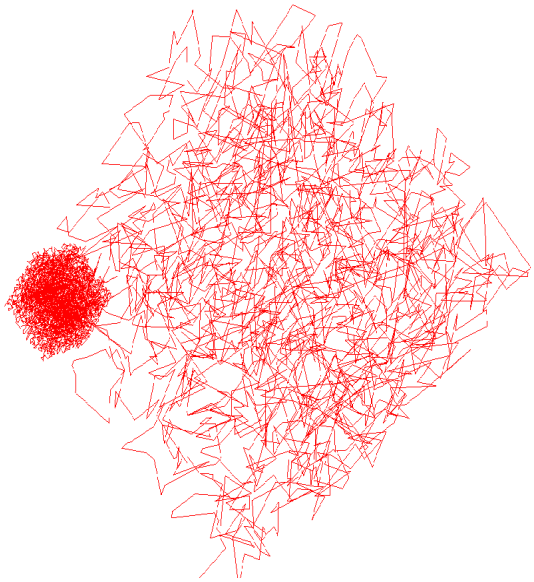


Figure 4.7: A 3D Space-filling Curve on Irregular Cluster

The pseudo-code for recursive SFC traversal is provided below. The pseudo-code in algorithm 4.4 is the top-level call that initiates SFC traversal of the kd-tree. The algorithm optimizes the curve if it can identify clusters, else defaults to the cis, $tran$ rules explained earlier. The pseudo-code in algorithm 4.3 invokes cis or $trans$ rules at

Algorithm 4.2 Node Resolution

```
1: procedure CHECK RESOLUTION(node * n)
2:     r ← 2
3:     d1 ← n.dim
4:     subcells ← level1_subcells(n)
5:     if ∀i ∈ [0, 2)subcells[i].leaf then return
6:     else
7:         if ∀i, j ∈ [0, 2)subcells[i].dim ≠ subcells[j].dim then return
8:         else
9:             if ∀i ∈ [0, 2)subcells[i].dim ≠ d1 then
10:                res ← 4
11:                d2 = subcells[0].dim
12:                subcells ← level2_subcells(n)
13:                if ∀i ∈ [0, 4), subcells[i].leaf then return
14:                else
15:                    if ∀i, j ∈ [0, 4]subcells[i].dim ≠ subcells[j].dim then return
16:                    else
17:                        if ∀i ∈ [0, 4)(subcells[i].dim ≠ d2 ∧ subcells[i].dim ≠ d1) then
18:                            res ← 8
19:                            d3 = subcells[0].dim
20:                        end ifreturn
21:                    end if
22:                end if
23:            end if
24:        end if
25:    end if
26: end procedure
```

Algorithm 4.3 CisTran Rules

```
1: procedure CISTRAN(node * n)
2:     split_dim ← n.dim()
3:     entry ← n.entry()
4:     exit ← n.exit()
5:     axis ← n.axis()
6:     if  thenopposite(split_dim, entry, exit)
7:         TRANS(n)
8:     else
9:         CIS(n)
10:    end if
11: end procedure
```

a node depending on whether the entry and exit faces are along adjacent or opposite faces. The algorithm 4.2 is used to identify clusters of 4 or 8 sub cells. At each node, we check the splitting dimensions of its sub cells at the next two levels of recursion if they are non-terminals. If all the sub cells at a particular level have the same splitting dimension and it is not equal to the dimensions at the previous levels, then, we increase the resolution r of the node to $2 * r$. r is initialized to 2 for a non-terminal node in the tree. If the resolution returned is 4, then we have identified a group of sub cells that lie in a plane. If the resolution is 8, then the sub cells lie at the corners of a cube, forming a cluster of 8. The pseudo-code in algorithm 4.1 is the method that invokes the 2D and 3D Hilbert rules if clusters were found in the kd-tree. Figure 4.5 shows GenSFC traversing a set of points that are equi-distant from each other. This could represent a finite-difference mesh where the SFC traverses the centroids of mesh elements. Figure 4.6 is the traversal of a uniform distribution of points in 3D, where point co-ordinates are random numbers in a volume ranging from $0 - 1000$. The figure 4.7 is a uniform distribution with a cluster in the lower left corner. GenSFC traverses this domain without large jumps where the distributions change. Transition from the cluster to the rest of the domain is smooth.

Algorithm 4.4 Recursive SFC Construction

```
1:  procedure TRAVERSE(node * n)
2:      if n.leaf then return
3:      else
4:          r ← CHECK RESOLUTION(n)
5:          if  then r == 8||r == 4
6:              subcells ← LOOKAHEAD OPTIMIZATIONS(n)
7:          else
8:              subcells ← CISTRAN(n)
9:          end if
10:     end if
11:     for  do i ← 1, subcells.size()
12:         TRAVERSE(subcells(i))
13:     end for
14: end procedure
```

4.4 3D SFC EMPIRICAL EVALUATION

In this section we present several test cases which address the problem of partitioning large datasets. The data is represented as a mesh of points, with elements and neighbors defined for each element. Empirical evaluation was done on the sequential implementation and not the parallel versions. The experiments in this section evaluate partitioning overheads (execution time and memory consumption) as well as the quality of partitions. All experiments in this section were performed on TACC's Stampede1 cluster. The compute nodes of Stampede1 contain two Intel Xeon E5 Sandy Bridge processors [84]. Measurements of partitioning overheads (memory and partitioning time) were conducted on a single Sandy Bridge compute node.

We used Metis with its default tuning parameters, i.e. randomized heavy edge matching algorithm for coarsening and about 20 iterations of greedy refinement [29]. Between SFCs GenSFC was compared with two implementations of Morton Order.

One of the implementations, generated bit-string keys by interleaving the binary values of node co-ordinates [93]. The co-ordinates of the centroid of a tetrahedron were converted to 64-bit strings by rounding floating point values. Concatenated key strings ($3 * 64 = 192$) bits wide, were sorted using STL sorting routines.

The other implementation was based on GenSFC. We built the kd-tree and performed its in-order traversal for sorting them by Morton order. All experiments in this section are sequential. We have used midpoint splitters along the dimensions of maximum spread.

4.4.1 Structured Meshes

Structured meshes were generated by fixing the number of elements in each dimension and the number of neighbors per element (referred to as stencil). We used symmetric and asymmetric structured meshes for these experiments, with 7 neighbors per element. To clarify, a 7-point stencil has one neighbor in each direction (+x,-x,+y,-y,+z and -z). A mesh is considered symmetric if it has the same extent in all dimensions, otherwise it is treated as an asymmetric mesh.

The first set of experiments measures the partitioning overheads of SFCs vs multi-

level schemes. Two meshes were used for this purpose, a 100x100x100 mesh with 1000000 elements and a larger 200x200x200 mesh with 8000000 elements, both having 7 neighbors per element. The graph in figure 4.8 contains a plot of the memory consumed during graph partitioning by Metis and SFCs for both meshes. Memory used by a partitioning algorithm includes storage space for the mesh dataset along with any auxiliary data structures required by the algorithm. For a given mesh, the memory consumed by Metis is a function of the number of partitions. The amount of memory used by SFC algorithms is independent of the number of partitions. Between the three partitioners, our implementation consumes more memory, has lower data movement, which resulted in lower partitioning time. The trade-offs between memory consumption, data movement and execution time become apparent as we increase the size of the dataset. Notice the difference in performance of the three algorithms for 200x200x200 mesh.

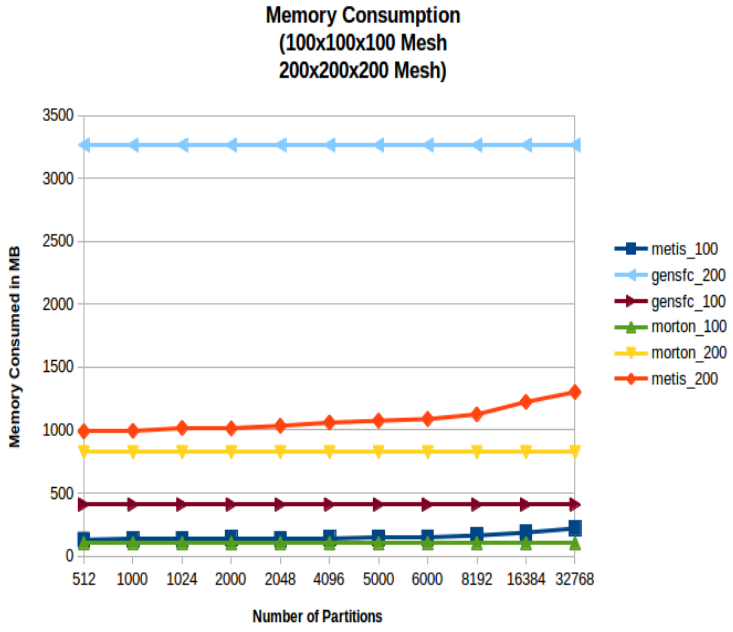


Figure 4.8: Memory Consumption of Metis partitioner for 100x100x100 mesh and 200x200x200 mesh

The quality of SFC partitions are comparable to Metis partitions for $100X100X100$ mesh, observations are recorded in tables 4.1, 4.2 and 4.3. Between Morton and GenSFC, our partitions have lower degree and communication volume. For meshes where elements have equal weights, SFC partitions guarantee load balance, i.e. the difference between maximum and average loads of a partition is at most one mesh element. For

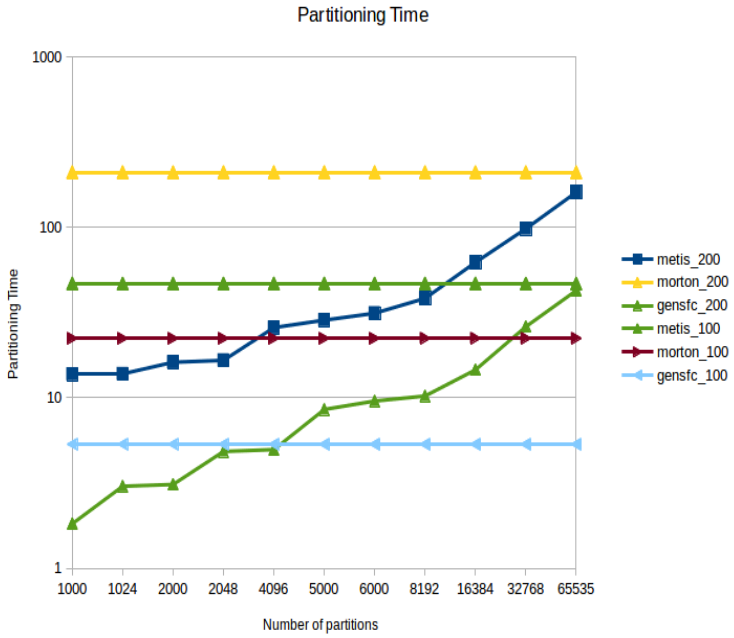


Figure 4.9: Partitioning Time for 100x100x100 mesh and 200x200x200 mesh

#cores	avg_load	max_load	max_deg	max_edge_cut
512	1953.125	1954	20	1286
1000	1000	1030	20	842
1024	976.5625	1005	20	838
2000	500	515	24	554
2048	488.28125	502	21	542
4096	244.140625	251	21	334
5000	200	206	22	302
6000	166.67	171	23	262
8192	122.0703	125	23	216
16384	61.035	62	23	142
32768	30.5175	31	22	91

Table 4.1: Metis Results for Structured Mesh (100x100x100) 7-point stencil

#cores	avg_load	max_load	max_deg	max_edge_cut
512	1953.125	1954	15	1380
1000	1000	1001	17	892
1024	976.5625	977	16	884
2000	500	501	16	574
2048	488.281	489	17	566
4096	244.140	245	17	314
5000	200	201	17	314
6000	166.67	167	18	282
8192	122.0703	123	19	232
16384	61.035	62	18	150
32768	30.5175	31	18	92

Table 4.2: Morton Results for Structured Mesh (100x100x100) 7-point stencil

#cores	avg_load	max_load	max_deg	max_edge_cut
512	1953.125	1954	12	1218
1000	1000	1001	15	828
1024	976.5625	977	14	800
2000	500	501	15	536
2048	488.281	489	15	516
4096	244.140	245	15	354
5000	200	201	16	310
6000	166.67	167	15	270
8192	122.0703	123	15	214
16384	61.035	62	16	150
32768	30.5175	31	16	96

Table 4.3: GenSFC Results for Structured Mesh (100x100x100) 7-point stencil

Metis, maximum load imbalance is tuned for the default value of 3percent.

Figure 4.10: Edge Cut of 100x100x100 mesh partitions

#cores	avg_load	max_load	max_deg	max_edge_cut
1000	8000	8001	15	3508
1024	7812.5	7813	17	3468
2000	4000	4001	15	2260
2048	3906.25	3907	16	2240
4096	1953.125	1954	17	1400
5000	1600	1601	17	1194
6000	1333.33	1334	18	1072
8192	976.56	977	17	892
16384	488.281	489	17	568
32768	244.006	245	17	360
65535	122.07	123	18	232
131070	61.036	62	19	150

Table 4.4: Morton Results for Structured Mesh (200x200x200) 7-point stencil

The results for $200X200X200$ mesh are presented in tables 4.4 and 4.5. Between the two space-filling curves, our algorithm produces consistently better partitions than Morton. Partitions have lower degree and edge-cut values for all process counts, including non-powers of 2, when gensfc partitions are spread across multiple subtrees of the kd-tree.

The third experiment in this category, uses an asymmetrical mesh with different extents in each dimension. We used an asymmetric mesh of size $200X100X150$. Results are recorded in tables 4.6 and 4.7.

The difference between Morton and gensfc partitions is more apparent when the

#cores	avg_load	max_load	max_deg	max_edge_cut
1000	8000	8001	15	3316
1024	7812.5	7813	12	2718
2000	4000	4001	15	2128
2048	3906.25	3907	12	1920
4096	1953.125	1954	13	1224
5000	1600	1601	16	1192
6000	1333.33	1334	16	1038
8192	976.56	977	16	800
16384	488.281	489	16	516
32768	244.006	245	16	354
65535	122.07	123	18	214
131070	61.036	62	17	150

Table 4.5: GenSFC Results for Structured Mesh (200x200x200) 7-point stencil

#cores	avg_load	max_load	max_deg	max_edge_cut
1000	3000	3001	15	1994
1024	2929.687	2930	15	1990
2000	1500	1501	16	1284
2048	1464.843	1465	16	1264
4096	732.421	733	16	780
5000	600	601	17	698
6000	500	501	18	596
8192	366.21	367	17	520
16384	183.105	184	17	342
32768	91.502	92	18	208
65535	45.77	46	18	134

Table 4.6: Morton Results for Asymmetric Structured Mesh (200x100x150) 7-point stencil

#cores	avg_load	max_load	max_deg	max_edge_cut
1000	3000	3001	16	1832
1024	2929.687	2930	15	1796
2000	1500	1501	17	1164
2048	1464.843	1465	16	1152
4096	732.421	733	16	720
5000	600	601	17	622
6000	500	501	18	560
8192	366.21	367	17	458
16384	183.105	184	17	302
32768	91.502	92	18	190
65535	45.77	46	18	120

Table 4.7: GenSFC Results for Asymmetric Structured Mesh (200x100x150) 7-point stencil

Figure 4.11: Edge Cut of 200x100x150 mesh partitions

meshes are asymmetric. The edge cuts of SFC partitions for $200X100X150$ mesh are plotted in figure 4.11. Morton order is defined on meshes of dimensions that are powers of two. For non-powers of two, the discontinuities in the curve resulted in partitions with higher degree and edge-cut.

4.4.2 Unstructured Meshes

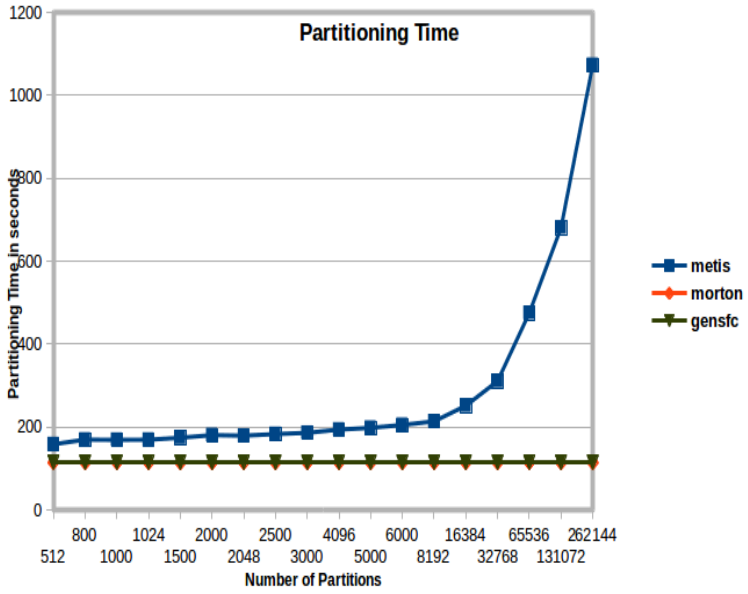


Figure 4.12: Partition time for unstructured tetrahedral mesh with 100745239 elements

We used tetrahedral meshes generated using Tetgen [94] for empirical evaluation. Uniform random distributions of points were provided as inputs to Tetgen, which constructed tetrahedral meshes by triangulating them (Delaunay methods). The Delaunay algorithm used by Tetgen has options to improve the quality of the mesh by refining it and adding Steiner points [94]. We used Metis as baseline for comparing quality of partitions. Each element has exactly four neighbors with which it shares a common face. We have included only face neighbors in our calculations for degree and edge-cut.

Input to space-filling curves is a set of points which correspond to centroids of mesh elements. Traversal of these points is a traversal on the tetrahedrons since elements are not split. When the curve is sliced, we get a partition of the mesh elements. To improve the partitioning time of space-filling curves for large meshes, kd-tree construction was

modified to reduce data copying and total memory consumption, described in the earlier chapters. The plot in figure 4.13 compares partitioning time for the two Morton order implementations for 50million elements.

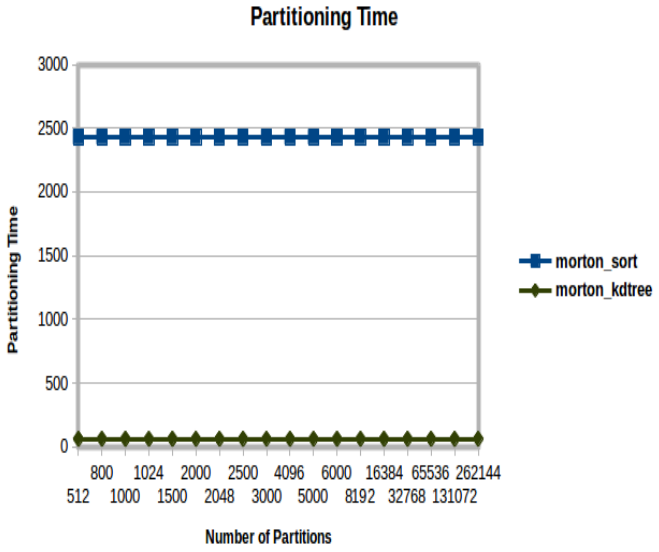


Figure 4.13: Morton order partitions for unstructured tetrahedral mesh with 51443520 elements

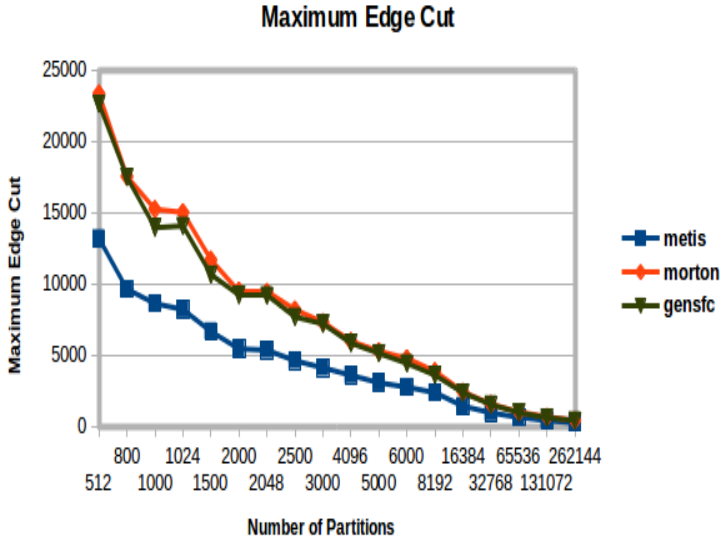


Figure 4.14: Edge Cut of unstructured mesh partitions (100m elements)

Partition qualities of SFC and Metis were compared for a tetrahedral mesh with 100million elements, shown in figures 4.14 and 4.15. Metis partitions had consistently lower edge-cut values. The measured values are tabulated in the tables 4.8, 4.9 and

#cores	avg_load	max_load	max_degree	max_edge_cut
512	196768.044	196769	33	23328
800	125931.548	125932	33	17528
1000	100745.239	100746	34	15232
1024	98384.022	98385	35	15008
1500	67163.492	67164	34	11696
2000	50372.619	50373	33	9528
2048	49192.011	49193	34	9438
2500	40298.095	40299	33	8198
3000	33581.746	33582	33	7322
4096	24596.005	24597	34	5986
5000	20149.047	20150	35	5286
6000	16790.873	16791	34	4780
8192	12298.002	12299	34	3880
16384	6149.001	6150	34	2440
32768	3074.5	3075	33	1576
65536	1537.25	1538	35	1022
131072	768.625	769	35	690
262144	384.312	385	36	442

Table 4.8: Morton Results for Unstructured Mesh with Tetrahedral elements

#cores	avg_load	max_load	max_degree	max_edge_cut
512	196768.044	196769	26	22610
800	125931.548	125932	26	17484
1000	100745.239	100746	26	13944
1024	98384.022	98385	26	14066
1500	67163.492	67164	26	10654
2000	50372.619	50373	25	9220
2048	49192.011	49193	25	9188
2500	40298.095	40299	26	7688
3000	33581.746	33582	25	7200
4096	24596.005	24597	27	5854
5000	20149.047	20150	27	5142
6000	16790.873	16791	26	4438
8192	12298.002	12299	26	3614
16384	6149.001	6150	26	2378
32768	3074.5	3075	27	1552
65536	1537.25	1538	27	982
131072	768.625	769	26	634
262144	384.312	385	27	432

Table 4.9: GenSFC Results for Unstructured Mesh with Tetrahedral elements

#cores	avg_load	max_load	max_degree	max_edge_cut
512	196768.044	201297	21	13190
800	125931.548	129145	22	9632
1000	100745.239	103181	21	8628
1024	98384.022	101164	21	8206
1500	67163.492	68944	23	6662
2000	50372.619	51707	22	5450
2048	49192.011	50502	22	5364
2500	40298.095	41441	22	4622
3000	33581.746	34507	22	4124
4096	24596.005	25334	22	3602
5000	20149.047	20752	22	3078
6000	16790.873	17295	26	2772
8192	12298.002	12667	25	2386
16384	6149.001	6333	27	1440
32768	3074.5	3166	27	974
65536	1537.25	1583	32	654
131072	768.625	791	33	436
262144	384.312	395	35	272

Table 4.10: Metis Results for Unstructured Mesh with Tetrahedral elements

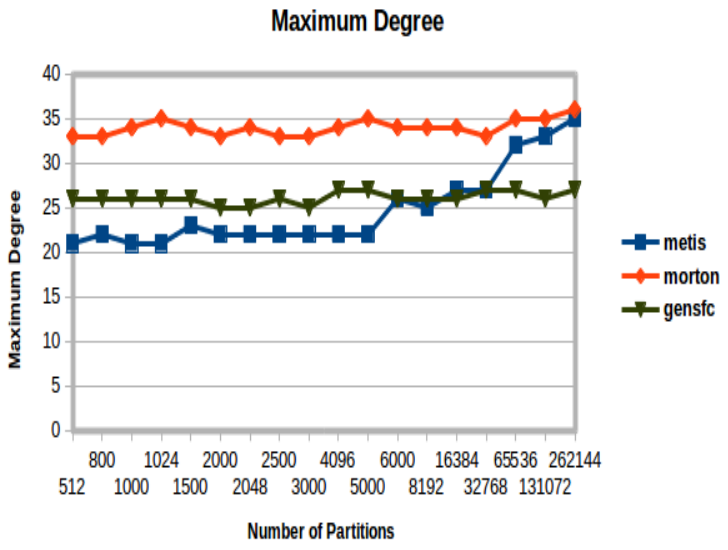


Figure 4.15: Max Degree of mesh partitions

4.10. Between the two space-filling curves, GenSFC has lower edge-cut and degree compared to Morton. This is due to better clustering and locality in our curves. The maximum degree of Metis partitions increased at large number of partitions. GenSFC had the lowest communicating neighbors for partitions $>= 6000$.

4.5 PARALLEL CONSTRUCTION OF GENERAL SPACE-FILLING CURVES

#threads	tree_time	trav_time	total
8	8.05007	0.25999	8.31006
16	8.08956	0.14347	8.23303
32	4.71578	0.0935593	4.8093393
64	2.48088	0.148985	2.629865
128	1.58521	0.105974	1.691184
256	2.86065	0.11795	2.9786

Table 4.11: Parallel SFC traversal time for 10million points, bucket size=32

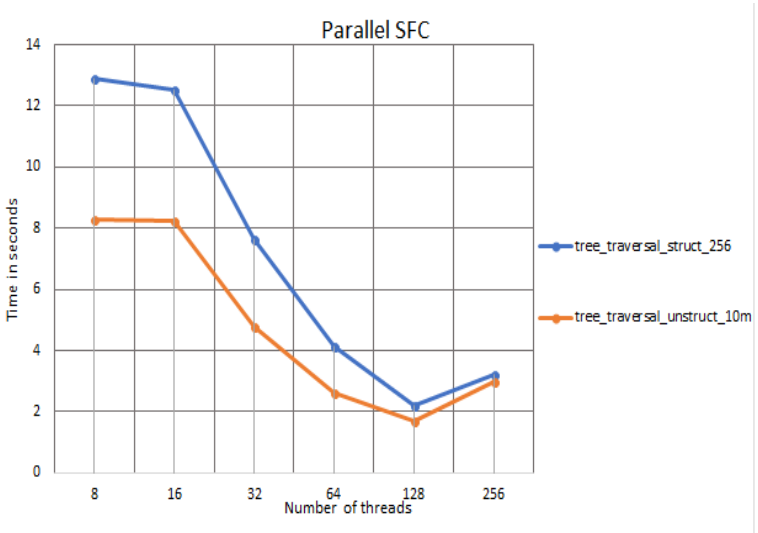


Figure 4.16: Parallel SFC on 256X256X256 mesh and 10m points, single-node performance

Hierarchical domain decomposition using kd-trees was discussed in detail in previous chapters, along with rules for a 3D space-filling curve to traverse them. In this section, we discuss the cost of constructing these curves in parallel.

Given $BUCKETSIZE = b$, the average number of leaves in the kd-tree is $\lceil \frac{n}{b} \rceil$. Assuming a balanced tree, the average depth of a tree with $\lceil \frac{n}{b} \rceil$ leaves is $log(\frac{n}{b})$. Each bucket or leaf node is visited at most once during traversal. Time taken to reach a leaf

node is equal to its depth in the tree, $log(\frac{n}{b})$. Average cost of sequential tree traversal, T_{trav} is :

$$T_{trav} = \frac{n}{b} * log(\frac{n}{b}) \tag{4.1}$$

Although the cost of traversal is less than the cost of tree-building, for large meshes or point distributions, traversal can become an overhead. Following along the lines of parallel tree-building, traversal costs can be reduced by allowing threads to traverse subtrees in parallel. The top nodes of the tree are traversed by a single thread, assigning directions($entry$ and $exit$) to each node. The number of top-nodes is $k*P, k >= 1$ where P is the number of threads. For the experiments in this thesis where $P >= 64$, we fixed the number of top-nodes to lie between $128 - 256$, 256 being the largest number of threads per MPI rank. In the next phase, top nodes are assigned to threads which build subtrees independently. After subtrees are built, a second pass traverses them in parallel, creating local orders for the leaf nodes owned by each thread. These segments are concatenated to create the final permutation of all leaf nodes in the tree.

Top nodes are traversed following a breadth-first order, assigning directions to coarse partitions. Subtrees are traversed in depth-first order.

There are dependencies in our SFC algorithm. The $entry\_pt$ of a box, depends on where the curve exited the previous box. For lower nodes of the tree which are traversed in depth-first order, this information is not available until the entire sub-tree is traversed. This would make the traversal algorithm sequential. To avoid this, directions and entry points are assigned to top nodes of the tree by thread0, after which these nodes are disconnected from each other. Each tree in the forest produces a segment of the final space-filling curve. There is no further communication between threads to determine directions in their portions of the SFC. A parallel-prefix on the number of points owned by each thread determines locations in the permutation array where local orders are copied.

One can further optimize the solution, by traversing sub-cells while the tree is being built. In the current version, traversal starts after the entire tree is built.

Assuming the assignment of top-nodes to threads is load-balanced, there are at most $\lceil \frac{n}{P} \rceil$ points per thread. Average number of leaf nodes per thread is $\lceil \frac{n}{P*b} \rceil$. Traversal cost is the maximum across P threads. For the load balanced case, average parallel

traversal cost is

$$T_{trav_p} = \lceil \frac{n}{P * b} \rceil * log(\lceil \frac{n}{P * b} \rceil) \tag{4.2}$$

#threads	tree_time	trav_time	total
8	123.926	38.0506	161.9766
16	121.983	40.9924	162.9754
32	95.5164	54.4375	149.9539
64	83.9269	56.55	140.4769
128	69.1433	47.69	116.8333
256	46.3058	30.811	77.1168

Table 4.12: Parallel SFC traversal time for 100million points, bucket size=100

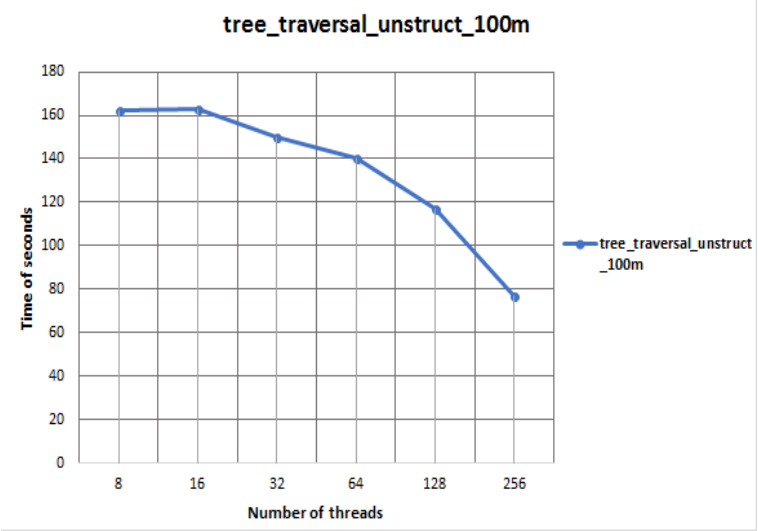


Figure 4.17: Parallel SFC on 100m points, single-node performance

4.5.1 Testcases

The parallel SFC implementation was tested on both structured and unstructured data in 3D. For the structured testcase, we used a regular grid, that is symmetric in all dimensions. The grid dimensions were $256X256X256$ and $BUCKETSIZE = 32$. The performance of parallel SFC includes both tree building and traversal times. We measured the performance of this implementation on a single KNL node, with different thread counts. For the testcase with unstructured data, we sampled points from a uniform distribution $[1, 1000000000]$. The SFC was used to traverse these points in par-

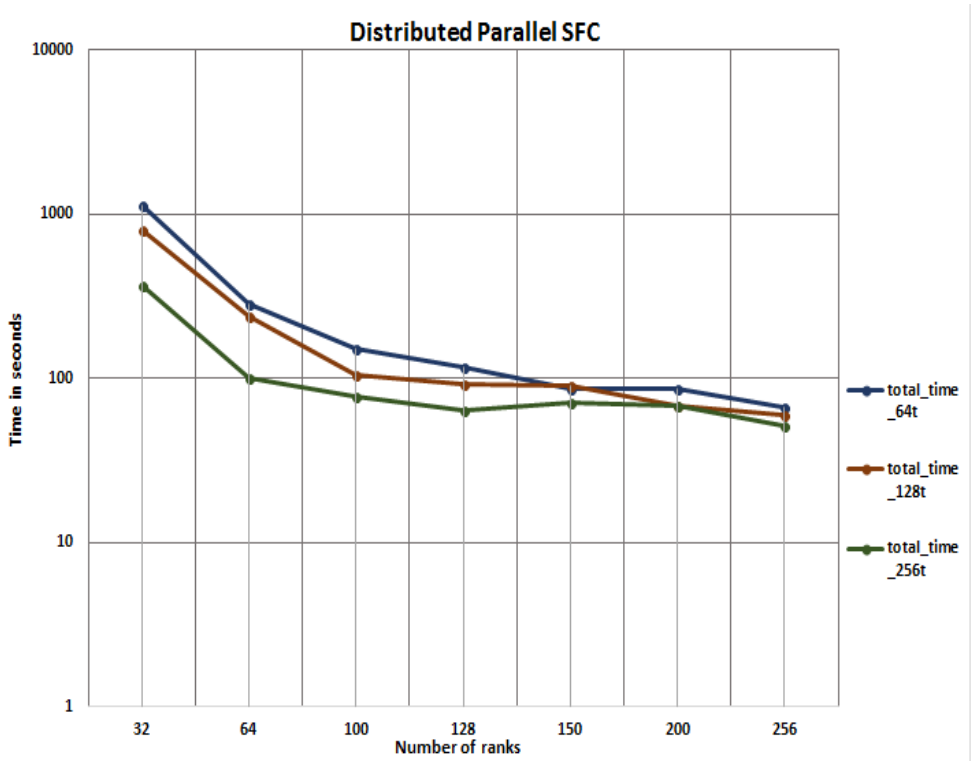


Figure 4.18: Parallel SFC on a uniform distribution of 8 billion points

allel. Although we have used points, they can also be centroids of mesh elements. The traversal does not depend on the shape of the mesh element - co-ordinates of centroids are sufficient. We have used two datasets - 10million and 100million points each.

The graph in figure 4.16 shows the performance of parallel traversal on a $256X256X256$ grid and $10m$ points. Number of threads ranges from 8 to 256. Both traversals seem to scale well with increasing thread counts. The graph in 4.17 shows the performance of the $100m$ points testcase for the same number of threads. These results are tabulated in tables 4.11 and 4.12. Besides shared memory implementations, parallel SFC traversal is also done on distributed kd-trees that are split across multiple processes. After constructing top nodes of the tree, they are assigned rules for traversal. Assignment of rules to top nodes is deterministic and therefore a replicated computation, performed at all processes. There is no communication involved in this step. Top nodes are distributed to processes in a load balanced manner and subtrees are built locally. For subtree traversal, we used the shared memory implementation described here. A parallel SFC traversal on 8 billion points is tabulated in table 4.13. The results are also plotted in the graph in figure 4.18. The values on the y-axis of graph 4.18 are based on log scale.

#ranks	th	topnodes	lbtime	subtree	init	travtime	total
32	64	13.5496	18.3857	815.362	0.0552484	277.092	1124.444548
64	64	7.31554667	12.6197	249.98367	0.033392033	14.7868	284.7391054
100	64	4.942162	9.04777	131.481	0.02884408	5.76071	151.2604861
128	64	4.290788	15.697054	92.66934	0.0282573	4.046772	116.7322113
150	64	3.625778	17.216508	62.96438	0.02017692	3.1467006	86.97354352
200	64	3.14199	26.43724	55.5043	0.03217158	0.634003	85.74970458
256	64	2.785552	26.79584	36.38254	0.03332384	0.38015788	66.37741372
32	128	23.155	27.7989	616.395	0.0596033	129.759	797.1675
64	128	14.0693	16.1688	186.635	0.0345392	21.5781	238.4857
100	128	9.6632	14.12866	76.2568	0.03096324	4.995312	105.0749352
128	128	8.160532	26.41452	56.06118	0.02689906	1.983812	92.64694306
150	128	7.30167	24.79238	57.51528	0.02695494	1.6494634	91.28574834
200	128	5.897814	30.05282	31.96828	0.02426034	0.5758874	68.51906174
256	128	5.237004	34.56682	19.73738	0.02584964	0.50309744	60.07015108
32	256	16.358	53.4606	236.004	0.0652721	59.9075	365.7954
64	256	8.98341	31.2315	52.4176	0.0435525	8.85853	101.5346
100	256	6.418946	23.20536	45.4614	0.04460024	2.924508	78.05481424
128	256	5.408766	33.33038	24.8794	0.04159662	0.454452	64.11459782
150	256	5.025954	37.23808	27.99508	0.04817824	0.5353426	70.84263484
200	256	4.306316	41.9783	21.42474	0.04149962	0.3287752	68.07963082
256	256	3.862756	36.50002	11.40896	0.04921726	0.219206	52.04015926

Table 4.13: Parallel SFC traversal time for 8 billion points, bucket size=200

CHAPTER 5: BENCHMARKS

In this section we discuss suitable testcases for space-filling curve partitions. There are several applications in scientific computing that use space-filling curves, especially Morton order [95], [96], [97], [98], [99], [100], [101], [102]. These are usually adaptive computations which require frequent re-partitioning, e.g [17]. We have used miniapps [103], which are stripped down versions of large applications, as testcases. These programs are benchmarks for analysing computation costs, communication costs or both in the applications that they model. Such benchmarks help in isolating and identifying potential bottlenecks in large applications. They are fertile environments for developing and testing new algorithms, programming models, etc. We used benchmarks to evaluate various overheads of load-balancing and data partitioning. Later, we used them for developing low overhead hybrid algorithms that involve both distributed and shared memories. The objective is to reduce the total execution time of parallel programs. These are the observations we have had from our experiments with benchmarks :

1. Reduce communication between processes by using better data decompositions.
2. Reduce load-balancing overheads by keeping the cost of load-balancing decisions to a minimum.
3. Reduce frequency of re-partitioning by identifying the best timesteps to load balance.
4. Reduce data migration during load balancing and re-assignment.
5. Re-write algorithms to suit the architecture of the processor.
6. Reduce synchronization and use low-overhead synchronization between threads in a process.
7. Use data-structures that favour parallelism e.g shared queues and hash tables vs shared stacks.
8. Use data-structures that favour spatial locality if the architecture has memory hierarchies.

The testcases used here have simple computation kernels as we were focused on load-balancing and communication costs. To model compute intensive kernels, com-

putation time can be replaced by a scaled constant factor α, which is the work per data point, or replace the kernel routines. For the default kernel, $\alpha = 1$. Default kernels compute the average of a measured parameter, over a pre-defined stencil. Testcases are iterative in nature, where computation and communication are repeated over many timesteps. We have identified two types of testcases based on workload distribution and its rate of change. Total computation work per partition is defined as the number of data points multiplied by the work per data point.

1. Static Workload - Total workload is constant in these applications. Load-balancing is performed once to distribute the workload across partitions. Processes perform computation and communication at every iteration/time-step without any changes to the workload. Poisson process [104] on a structured mesh is an example for static workload. There are two ways in which computation and communication can be arranged - blocking kernel and non-blocking kernel, shown in algorithms 5.1 and 5.2. A blocking kernel performs computation and communication in stages. Threads synchronize at a barrier until the computation phase is over. They enter communication routines together, exchange boundaries and update their blocks with new data from neighbors. In these kernels threads are idle until remote messages are received. A faster approach in some cases is to overlap computation and communication. If using MPI for communication, non-blocking messages containing boundaries are posted before computation. Once the computation step is over, threads wait for communication to complete. The messages received are used to update boundaries of blocks. Processes require additional buffers to receive and save incoming data when there is overlap. If processes synchronize between computation and communication stages messages can be ordered and partial results accumulated without allocating extra memory to store messages from remote neighbors. There is a trade-off between these two approaches. We pick one over the other depending on the number of parameters observed and total message size. For large messages, if the communication cost is predominant, overlapping the two may be a better option. In our testcases, blocksizes are non-trivial and there are 10 observed parameters at every point. We have overlapped communication and computation stages in the stencil routine (referred to as kernel here) for all testcases. For other code sections, we have used the blocking model, e.g mesh refinement. Refinement algorithms are explained in detail later, but to be brief, there is very little computation involved in these methods. They are communication-based, with short messages exchanged

until consensus is achieved. The pseudo-codes below show the difference between these two kernels in the way they are programmed.

Algorithm 5.1 Blocking Kernel

```
1: procedure BLOCKING_KERNEL(ts)
2:     COMPUTATION_STEP
3:     COMMUNICATION_STEP
4:     BOUNDARY_COMPUTATION
5: end procedure
```

Algorithm 5.2 Non-Blocking Kernel

```
1: procedure NONBLOCKING_KERNEL(ts)
2:     COMMUNICATION_BEGIN
3:     COMPUTATION_STEP
4:     COMMUNICATION_END
5:     BOUNDARY_COMPUTATION
6: end procedure
```

2. Dynamic Workload

   Workloads can be dynamic in two ways :

   (a) Total size of the mesh changes over time by the addition and deletion of points to the domain. This may be due to changes in the underlying physics that the mesh models, e.g : turbulence. More points are sampled in areas of rapid change, while fewer points need to be sampled from stable regions [105]. The sampling rate may depend on the simulation timestep as well as the parameters being monitored. Some parameters change faster than others. All these criteria lead to different snapshots of the domain with regions of clustering and sparse data. Each version of the mesh has different statistics - total mesh size, number of coarse cells, number of fine cells, maximum number of cell neighbors etc.

   (b) Shape of the mesh may change over time ,e.g moving dataset in collision detection problems [106], climate simulation [99], tracking problems [67]. The total number of points may or maynot change. Load balancing for these problems extends to changes in communication workload. We have parameterized communication cost using two metrics - maximum number of messages and maximum communication volume between any two processes. Changes in shape may affect the total execution time by producing partitions with large boundaries, although the computation cost is load balanced.

These problems may or maynot require frequent load balancing depending on the application, i.e its ratio of communication cost vs computation cost. We have provided an option for the programmer to request full re-partition even if the computation cost is balanced between threads.

An example for programming adaptive meshes with dynamic workloads is provided below. There are two versions provided here - blocking and non-blocking kernels.

Algorithm 5.3 Blocking Kernel with Dynamic Workload

```
1:  procedure BLOCKING_KERNEL(maxts)
2:      for do ts ← 1, maxts
3:          if then ts%refine_freq == 0
4:              REFINE_COARSEN
5:              delta ← LOAD_IMBALANCE
6:              if then delta > threshold
7:                  REPARTITION
8:              end if
9:          end if
10:         COMMUNICATION_STEP
11:         COMPUTATION_STEP
12:         BOUNDARY_COMPUTATION
13:         ts ← ts + 1
14:     end for
15: end procedure
```

All the testcases we used belong to the second category - dynamic workloads. We have developed our own benchmarks for adaptive mesh refinement, based on **MiniAMR**, from Sandia. The kernels described in algorithms 5.3 and 5.4 are descriptions of our implementation. The $refine\_coarsen$ step marks blocks for refinement, based on a marking function. The base implementation generated different patterns for marking. We used some of them in the sequential version for the sake of comparison. Otherwise, our marking function generates a set of random points for refinement in the next timestep. Mesh points are aggregated into blocks, where each block has a refinement level associated with it. In this thesis we discuss refinement that is subject to one constraint : adjacent levels differ by at most one level. The refinement algorithm has a domino effect, where refinement of a block can trigger the refinement of other blocks in the neighborhood to satisfy level constraints. The algorithm terminates at steady state, when all blocks have satisfied their level constraints. Out of all remaining blocks, those

Algorithm 5.4 Non-Blocking Kernel with Dynamic Workload

```
1: procedure NONBLOCKING_KERNEL(maxts)
2:     for do ts ← 1, maxts
3:         if then ts%refine_freq == 0
4:             REFINE_COARSEN
5:             delta ← LOAD_IMBALANCE
6:             if then delta > threshold
7:                 REPARTITION
8:             end if
9:         end if
10:        COMMUNICATION_BEGIN
11:        COMPUTATION_STEP
12:        COMMUNICATION_END
13:        BOUNDARY_COMPUTATION
14:        ts ← ts + 1
15:    end for
16: end procedure
```

which can coarsen, will be replaced. Although refinement check is performed every $refine\_freq$ steps, addition and deletion of blocks may happen at a slower pace depending on how the simulation evolves. This is especially true in the case of simulations that model natural phenomenon [105]. Refinement rate may be much higher for other simulations like car simulations [106] and [107]. A simple implementation of the $load\_imbalance$ routine computes average and maximum number of blocks per thread/process. Differences between maximum and average number of blocks is $delta$, which is the measure of load imbalance. The repartition phase reassigns blocks to threads/processes following some pre-defined order, usually Morton or Hilbert. One can choose an order that minimizes degree and edge-cut of the communication graph, without increasing the cost of re-partitioning and data migration.

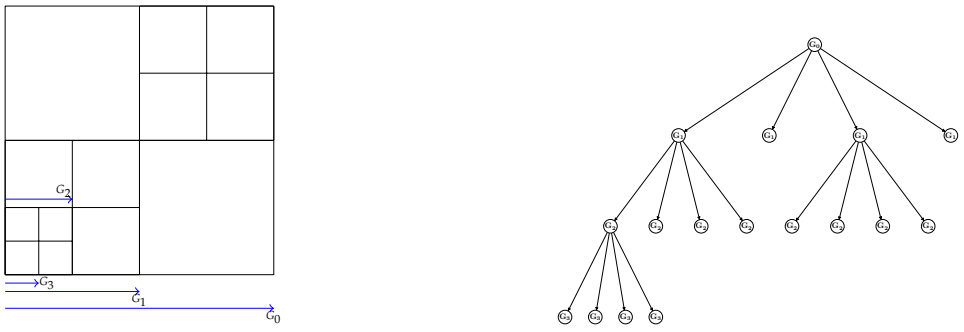
5.1 MINIAMR - BLOCK-STRUCTURED AMR

MiniAMR focuses on octree meshes which are block-structured. For a given level of refinement, mesh points are sampled from a domain with uniform resolution in all dimensions. Let the resolution be r at level l, level $l+1$ has resolution $\frac{r}{2}$ and level $l-1$ has resolution $2 * r$. Points are grouped into axis-parallel hypercubes, the dimensions

of which are configurable. All blocks have the same blocksizes, i.e the same number of points in all dimensions. Blocks at different levels cover different volumes in the physical domain, but they contain the same number of points. If a block at level l encloses a volume v, the same block at level $l+1$ covers $\frac{1}{8}^{th}$ the volume v. Conversely, when the block is at level $l-1$, it covers 8 times the volume v. Since the volume per block is not uniform, these meshes are considered separate from structured grids. The purpose behind using blocks is aggregation of nearby points, which gives spatial and temporal locality in the computation kernels, depending on block and cache sizes. Each block has a *halo* region defined around its mesh points, which is used for exchanging boundaries with neighboring blocks, irrespective of whether they are remote or lie within the same process. Blocks constituting a mesh are susceptible to refinement and coarsening operations.

1. Refinement : Splitting of a block into 8 new blocks.
2. Coarsening : Combining a group of 8 blocks into a single block.

Both refinement and coarsening should maintain 2:1 balance between neighbor blocks, where no two neighboring blocks differ by more than one refinement level. The miniapp uses a 7-point stencil with at least one neighbor in each direction, except at the boundaries. Neighbors are defined at block surfaces. The maximum number of neighbors in any direction is four. Relationships between refined and coarsened blocks are maintained using an octree. The mesh is present only at the leaves of the tree - it is the union of all leaf nodes. Therefore, kernels are executed at leaf nodes. From the perspective of trees, mesh refinement is better understood as tree manipulations that either remove a sub-tree of 8 leaves or add a new sub-tree of 8 leaves.



(a) Adaptively Refined Mesh (b) Quadtree

Figure 5.1

The base implementation from Sandia has a distributed memory model with MPI for

inter-process communication. The program per process is sequential, with one thread per MPI rank. The initial mesh can be a saved one from some previous simulation or created by the user. At least one block is placed per process to initialized it. Rest of the mesh is generated during simulation by continuous refinement and coarsening. The dual graph of this mesh is defined using leaf blocks as vertices and their neighbors as edges. During the simulation , blocks are refined, coarsened and data is exchanged between neighboring blocks. Workload is reassigned at fixed intervals for load balancing. Typical overheads in this program are the following :

Computation overheads :

- Storage and accesses : The storage of mesh blocks themselves can become cumbersome if block sizes are large.
- Meta-data representation and accesses : The mesh meta-data describing relationships need to be stored/represented efficiently.

Every operation in the benchmark accesses one or both of these data-structures. It is important that these data structures are fast and efficient and allow quick updates which may increase/decrease their sizes and modify access patterns.

Communication overheads :

- Refinement propagation: The refinement of a node may require refining adjacent nodes, in order to maintain the balance condition. These operations require communication between a node, its siblings and its neighbors.
- Halo-exchange: Blocks exchange boundary data with their neighbors at every time step of the simulation. Although overheads are different for each kernel type, all variants benefit from better load balance and data locality.

The overheads from a parallel adaptation are load balancing and re-partitioning. These depend on the data decomposition, assignment of tasks to threads/processes and mapping to cores. The most expensive part of load balancing is data migration to new threads/processes, which results in communication (including cache misses).

An entire simulation of MiniAMR is divided into well-defined stages, consisting of refinement and coarsening, stencil computation, nearest neighbor boundary updates and load balancing. Here we discuss briefly the different stages and their contributions to the total execution time of the simulation.

Let N_{ij} be the number of blocks located at process j at timestep i and P be the number of processes/threads. Let C be the computation cost per block. The simulation is initialized in a load balanced configuration. Let ref_freq be the refinment frequency. Assume a model where the kernel is blocked, i.e computation and communication are performed in stages. Let T_{comp_i} be the computation and T_{comm_i}, the communication components of the stencil phase at timestep i. T_{ref_i} and T_{lb_i} are the refinement and load balancing costs respectively at step i . We use the following notation; $T_{lb_{ij}}$ is the load balancing cost measured at process j during step i. The first index is the timestep and the second index is the process id.

Let $maxts$ be the total number of timesteps executed by the simulation. Processes and threads synchronize after refinement, stencil and load balancing stages. The costs per timestep are summed over the total number of timesteps and the frequency with which each of these routines are invoked by the application.

The stencil stage is executed in all timesteps. Let T_{sten} be the maximum time spent in the stencil routine by P processes accumulated over all timesteps. Similarly, T_{ref} is the total refinement cost for r refinements where $r = \lceil \frac{maxts}{ref\_freq} \rceil$ and T_{lb} is the total load balancing cost over k load balancing steps. We have used a PRAM delay model for communication where α is the latency and β the cost of transferring a byte across the network. Let d_{ij} be the number of messages and m_{ij} be the number of bytes sent/received by process j at timestep i.

$$T_{comp_i} = \max_{j=1}^{P}(C * N_{ij}) \tag{5.1}$$

$$T_{comm_i} = \max_{j=1}^{P}(\alpha * d_{ij} + \beta * m_{ij}) \tag{5.2}$$

$$T_{sten_i} = T_{comp_i} + T_{comm_i} \tag{5.3}$$

$$T_{sten} = \sum_{i=1}^{maxts} T_{sten_i} \tag{5.4}$$

$$T_{ref_i} = \max_{j=1}^{P} T_{ref_{ij}} \tag{5.5}$$

$$T_{ref} = \sum_{j=1}^{r} T_{ref_j} \tag{5.6}$$

$$T_{lb_i} = \max_{j=1}^{P} T_{lb_{ij}} \tag{5.7}$$

$$T_{lb} = \sum_{j=1}^{k} T_{lb_j} \tag{5.8}$$

$$T_{exec} = T_{sten} + T_{ref} + T_{lb} \tag{5.9}$$

For non-blocking kernels, we replace the stencil cost at step i with the following equations. $T_{sten_{ij}}$ is the cost of stencil computation and communication on process j at step i.

$$T_{sten_{ij}} = max(C * N_{ij}, \alpha * d_{ij} + \beta * m_{ij}) \tag{5.10}$$

$$T_{sten_i} = max_{j=1}^{P} T_{sten_{ij}} \tag{5.11}$$

This is also the model we have used for measuring time in our testcases. The synchronization points in the program closely follow this model. The total execution time should be the sum of the values for each phase.

In the next sections of this chapter, we discuss our implementations in detail.

5.2 MINIAMR IMPROVEMENTS

Although partitions have good empirical performance, sometimes they do not translate well into measured execution times. These are two possible reasons why the gap exists between predicted models and measured performance.

1. Datastructures without spatial locality. This was an issue in the base version that made it difficult to analyse the dependence of total execution time on load

imbalance. Most of the execution time was spent dealing with cache misses.

2. Poor algorithm design. Algorithms with high computation and communication complexities will contribute more towards the measured execution time. It may be difficult to isolate the contributions from load imbalance and kernel execution time in such scenarios.

Considerable time was also spent on developing a light-weight load balancing schemes. Load balancing at fixed intervals without examining the load distribution is usually wasteful.

The base version from Sandia is publicly available for download. This benchmark was modified in stages. All optimizations and their benefits/trade-offs are discussed in detail in the following subsections.

5.2.1 Optimization 1 : Block data structure

The base implementation uses an array of pointers for blocks. Blocks may be allocated anywhere in memory, therefore all memory accesses in the program were irregular, without any spatial locality. We have used flat arrays to store mesh elements and their data. The mesh meta-data (octree) is not stored explicitly, instead it is maintained using references to mesh elements by storing their keys. The mesh data structure has two levels :

1. Dictionary for locating mesh elements, neighbors, sibling and ancestors - the octree links.

2. Mesh data is stored in contiguous blocks. Assuming symmetric blocks, each block has a local co-ordinate system with dimensions ranging from $0 - (b - 1)$, where b is the blocksize. Blocks themselves have centroids derived from their positions in the global mesh. Global-to-local address translation is computed using the dictionary. When a neighbor block with centroid (x, y, z) is located in the mesh, we are indirectly accessing points in the range $[x - b/2, x + b/2), [y - b/2, y + b/2), [z - b/2, z + b/2)$.

Each block in the AMR tree has a unique key associated with it, that is derived from its relative position in the octree. The dictionary uses these keys as hash functions to

locate the mesh block in memory. If blocks need to be re-ordered for better cache re-use, SFC keys can also be used as hash keys. One can define various traversals on the AMR tree to map keys to blocks. The only criteria here is to keep block relationships intact. Some mappings are better than others at maintaining spatial locality in block accesses. For good mappings, keys assigned to blocks derived from the same coarse block should be numerically closer than keys assigned to neighboring subtrees. This is natural for SFC orders. We have used both Morton and Hilbert space-filling curves for key assignment.

All keys local to a process are stored in the dictionary. For typical problem sizes, the dictionary is small enough to fit in lower level caches where it can be re-used. Every access to a block, first looks for its key in the dictionary to identify the location of the block in memory. Although each block access requires at least two reads, this did not create additional overheads for us, because of cache re-use. It also provided a natural mechanism for global-to-local address translation and vice-versa. There is no need for explicit communication to locate non-local neighbors [93], [95]. Keys within a dictionary are arranged according to the order of blocks in memory. In our implementations, this is usually some space-filling curve order. If the dictionary is a vector of size n, at most $O(n)$ accesses are required to locate a block. In our sequential implementations, we have used binary search to locate keys. This reduced the average access time to at most $log(n)$ per block. We used a balanced binary tree implementation from Boost to store local keys [108].

A diagram showing the two-level data structure is provided in figure 5.2.

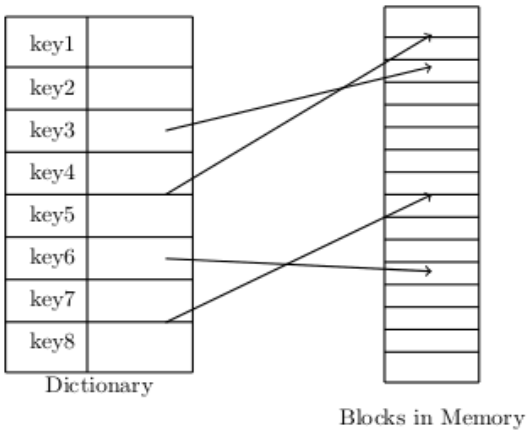


Figure 5.2: Two level Mesh Datastructure

In the sequential version, blocks are not arranged in memory in SFC order.

5.2.2 Optimization 2 : Refinement/Coarsening Algorithm

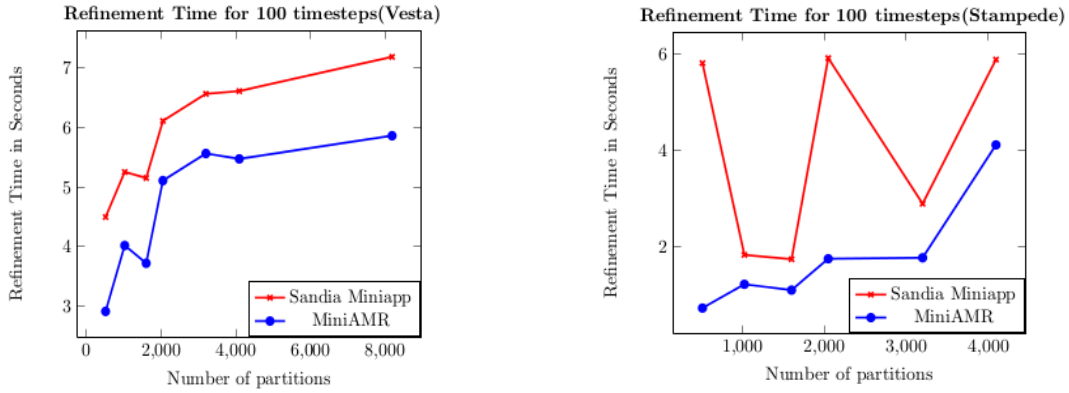


Figure 5.3: Test case 1 on Vesta and Stampede for Refinement and Coarsening

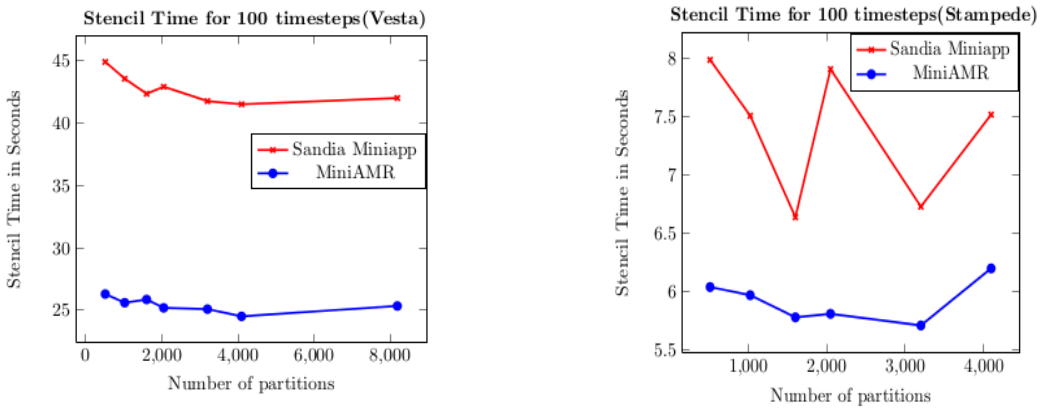


Figure 5.4: Test case 1 on Vesta and Stampede for Stencil Computation

Each AMR block is in one of three states - *refine*, *coarsen* and *stay*. All blocks are marked for coarsening by default. Some blocks in the domain may be marked for refinement or forced to stay at the current level. These blocks propagate their state information to siblings and neighbors. When blocks receive state updates from siblings and neighbors they store these values locally and use them to update their own states. For every refinement phase, this algorithm runs iteratively until a steady state is reached when there are no more transitions. If all siblings can coarsen, the entire group of 8 subcells is replaced by a single coarse cell, with updated neighbors and datapoints. Blocks update their neighbors keys depending on the state information received. The base implementation uses a version that involves both leaf nodes and non-terminals for decision making. Each leaf node sends its state to its parent node

which decides the final state of its children. The algorithm is iterative and terminates when there are no more messages exchanged between blocks. This implementation suffered from high communication costs. The block decomposition of leaf nodes was entirely different from that of non-terminals. This irregular communication lead to a slow implementation. Moreover messages were not aggregated. The maximum degree and edge-cut of the communication graph were very high. We had to re-design the algorithm for these reasons.

A pseudo-code for the improved refinement/coarsening algorithm is provided below:

The algorithm has two phases: the *Consensus* phase and the *Addition-Deletion* phase. Our implementation is similar to the prioritized ripple propagation algorithm in [109], but we have lower overheads due to better data layout and nearest neighbor communication. Besides, the algorithm in [109] does not allow blocks to coarsen.

1. Consensus

   This phase is iterative and repeated until quiescence.

   Each process maintains a local queue of blocks marked for refinement or forced to stay at the current level. A single iteration of the consensus algorithm performs the following steps :

   (a) Process entries in the local queue until it is empty.

   (b) Update states of neighbors that are local.

   (c) Aggregate messages to non-local neighbors in message queues. A message queue is maintained for every neighbor in the communication graph.

   This communication is highly localized and can be executed concurrently for different regions of the mesh since the *region of influence* of a block is limited. We make two assumptions here, that the input mesh is balanced and a block changes by no more than one level during a single refinement phase.

   The messages aggregated by a process during an iteration are exchanged using Sparse Collective routines [110] in MPI. This exchange of states could trigger further refinement of blocks, which are added to local queues for processing in the next iteration. The consensus algorithm terminates when local queues and message queues on all processes are empty. This state is defined as quiescence: when

all processes in the system are idle. We check for quiescence at the end of every iteration of the consensus algorithm. A pseudo-code for the algorithm is provided below (Algorithm 1).

Algorithm 5.5 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(quiescence)
24:     end while
25: end procedure
```

There are non-iterative versions of the parallel-consensus algorithm [111], [112]. However, in practice we found our algorithm converges quickly since refinement and coarsening are localized operations.

2. Addition/Deletion of blocks

   Refined blocks are added to the end of the block array without preserving SFC key order. Blocks that are marked for deletion during coarsening are removed and the holes are filled by shrinking the array. A new dictionary is created after this phase, because block locations have changed. This phase is relatively more expensive because blocks and data need to be copied in memory. We compared the effects of our optimizations with the base version from Sandia on two ma-

chines - Vesta, which is a supercomputer at ANL based on IBM powerpc nodes and Stampede1, a supercomputer at TACC, based on Intel Sandybridge nodes.

The graphs in figures 5.3 and 5.4 show differences in peformance between baseline Sandia version and our version with two optimizations enabled. This test case models an immersed moving sphere with refinement along its surface. The refinement frequency was set to 3 time steps, although addition and deletion of blocks took place every $6-9$ timesteps.

The version of MPI which has optimized on-node communication, Nemesis [113] was used on both machines. We placed one MPI rank per core. All results are reported for number of cores and not nodes. The experiments in this section show weak scaling results with approximately 300 blocks per process. Each block has exactly $4\times4\times4$ grid cells and 10 variables per point. The graphs in figure 5.3 show the total refinement time for 100 time steps on both machines. Figure 5.4 shows the total stencil time for 100 timesteps. The compute nodes and interconnection networks on these machines differ greatly. Hence, the comparisons are relevant and give valuable insights into useful program optimizations for each machine.

5.2.3 Optimization 3 : Load balancing

The frequency of load balancing is a critical determinant in the total execution time of the simulation. In the extreme case, one could trigger load balancing after every refinement/coarsening or have no load balancing at all. The base version partitions leaf nodes using a recursive bi-partitioning algorithm which is executed every lb_freq steps. Frequent load balancing creates good partitions at the cost of high partitioning overheads. In order to balance the two competing needs, we introduce the idea of *Amortization* [114] to load balancing. The cost of a load balancing phase (including data migration) is amortized over subsequent refinement and stencil phases. Formally, split the computation into segments of ref_reg iterations. If ref_freq is the frequency with which the refinement algorithm is invoked, each ref_reg segment consists of k subsegments, where each subsegment has ref_freq timesteps with stencil invocations followed by one timestep that includes a refinement phase. The load balancing routine is invoked at the end of a ref_reg segment if it satisfies the following cost model. Suppose the last load balancing invocation was at the end of timestep t_0, we number

this as subsegment 0; let C_{lb} denote the load balancing cost and let $T(t)$ be the time taken for the execution of subsegment t.

The next load balancing phase will be at the end of subsegment k where t_k is the first timestep that :

$$C_{lb} \leq \sum_{t=1}^{k} (T(t) - T(1)) \tag{5.12}$$

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

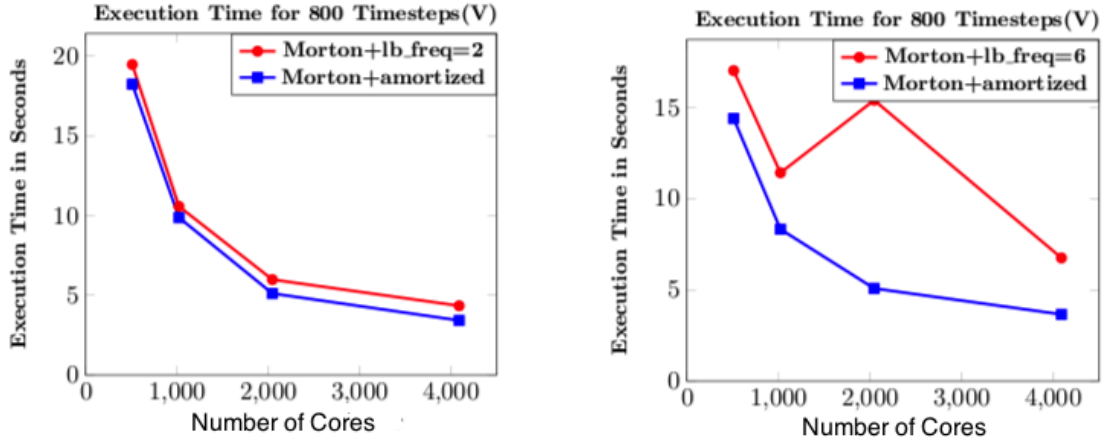


Figure 5.5: Test case for amortized load balancing on Stampede1

The graphs in figure 5.6 show the execution times obtained for a simulation with 800 time steps on Stampede1.

Both test cases had an initial domain covered by $64 \times 64 \times 64$ blocks and created a mesh with $220,000$ blocks ($14,080,000$ points). The explosion started as a point located at the center of this domain. The base case performs load balancing once every 6 time steps. The difference between test cases a and b is in the rate at which the explosion front expands. This affects the rate of change of computational load in the domain. Test case a evolves slower than test case b. In the case of test case b, frequent load balancing resulted in higher data migration costs. Therefore, we obtained better results with amortized load balancing for this experiment. To benefit from amortization, the load balancing overhead should be low compared to the other phases, including decision making and reassignment costs. This will allow the amortization equation to quickly detect variations in load across partitions.

The cost function C_{lb} can detect not only changes in load, but also changes in the total cost, including overheads from bad communication patterns (high degree and edge-cuts). The implementation of load balancing is relatively cheap. Every MPI rank sorts and computes the ranks of its local keys on the global SFC. Ranking blocks using SFC keys is a simple operation. An inclusive scan is performed on the sorted SFC keys, followed by local ranking. After ranking keys, the SFC is sliced into equal length segments. AMR blocks are packed and transferred to processes that own their keys.

5.2.4 Optimization 4 : SFC Partitions and Splitters

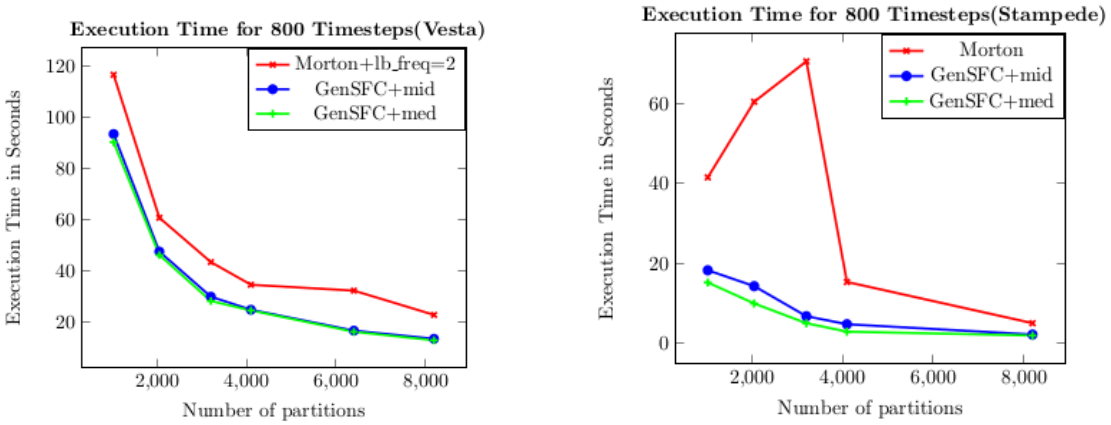


Figure 5.6: Clustered mesh on Vesta and Stampede1

Another optimization is the use of space-filling curve keys to order AMR blocks and the parallel generation of block keys. When new blocks are added during refinement, the keys for refined blocks are generated by concatenating three bits to the parent key, according to some space-filling curve. Position of the new block on the SFC is determined by the location of its key in the sorted key list. This technique was explored in other AMR packages like [101]. We ran the simulation for 800 timesteps with incremental SFC refinement when new blocks were added and deleted. Experiments were performed using two space-filling curves, Morton and GenSFC. For regular meshes, there was no difference between the execution times. But for clustered meshes, gensfc partitions had lower maximum degree and communication volume. The test case used for comparing curves, models turbulent flows by simulating the slow movement of a dense cluster of points. This test case has much more clustering than others. We created a slightly asymmetrical domain which is longer in the X dimension than Y and Z

dimensions. Our domain size is $32 \times 16 \times 16$ and we used 5 levels of refinement. The cluster is modeled using a sphere located at the lower left corner of the pipe. It slowly moves along the x direction as the simulation evolves. Refinement occurs throughout the volume of the sphere instead of just the boundary. The SFC is skewed with most of the load lying in the volume of the sphere. This leads to irregular communication since the few partitions that cover the remaining geometry of the domain communicate with those covering the cluster and end up as hotspots in the communication graph. They have very high degrees compared to the other partitions which lead to increased refinement and stencil time. The Morton order we used had a midpoint splitter. We used two versions of GenSFC, with midpoint and median splitters. The median splitter generated partitions with uniform communication degree and edge-cut. This test case reached a final mesh size of 500000 blocks (32000000 cells) both on Vesta and Stampede1. The results shown here are over a small simulation window of 800 time steps where the position of the cluster does not vary significantly. We used amortized load balancing for GenSFC partitions.The use of amortization and a median splitter reduced the execution time of the simulation by a maximum of 49% and an average of 32% on Vesta. We were able to improve the performance of this test case on Stampede1 by a maximum of 90% and an average of 69%.

5.3 MULTI-THREADED ADAPTIVE MESH REFINEMENT

This section discusses the distributed and multi-threaded versions of our AMR benchmarks. Baseline code is our distributed version discussed in the previous section, with one thread per MPI rank. We have not compared against the Sandia implementation in this section. All experiments were carried out on Stampede2 and measurements are consistent with the equations in 2.2. Introduction of multi-threading to the benchmark required considerable programming effort in terms of understanding dependencies in the algorithms as well as architecture of many-core nodes. All code sections were parallelized using threads, except the communication routines.

We adopted a SIMD [78] programming style for this program for the following reasons :

1. The baseline implementation of AMR has well-defined synchronization points where it is necessary for active threads to halt and collaborate before proceeding

with any further computation

2. All experiments were carried out on Intel KNL nodes, which are many-core with at most 8 NUMA nodes and 4 NUMA regions. Each KNL node consists of 34 tiles, two cores per tile and 4 hardware threads per core. All the cores on a tile share 1MB L2 cache. It was important to ensure that none of the cpus on the node are idle at any point during execution. Also, we took extra care to load balance workload between threads, and maximize cache re-use by mapping nearest neighbors to the same tile.
3. Construct testcases for hierarchical partitioning, enabled by space-filling curves.
4. Build a benchmark with little thread synchronization and scheduling overheads, make the algorithms contention-free and use lock-free data structures if necessary. An important observation from the previous chapters is to reduce repartitioning overheads, include amortization and other load-balancing schemes in the AMR simulation.

5.3.1 Memory Model

The approach used for multi-threading is similar to that described for kd-trees. Posix threads were used as sequential units of execution, where each thread has thread-local parameters, variables, as well as pointers to shared data structures. Co-ordination between threads is achieved with shared-memory mail-boxes, i.e by reading and writing from shared locations. There are no guarantees on the memory model unless specified by the programmer. Shared variables which are multi-reader and multi-writer are declared as atomic with a memory consistency model selected by the programmer. We used sequential consistency [115], [116] for all atomic variables in our programs, which were mainly shared counters, pointers in linked lists and block addresses so that the view within a thread is sequential for such variables. The consistency model for other locations with mutually exclusive accesses, including multi-reader locations is left unspecified. For multi-reader memory locations, explicit memory fences were used to ensure loads return the most recent values [117]. Three basic synchronisation primitives are used in these programs - reductions, prefixes and barriers. This has been explained in detail in the previous section along with performance numbers for each primitive.

The refinement/coarsening and load balancing algorithms went through several changes during the transition from sequential to multi-threaded. Stencil computation was straight-forward to parallelize. Similar to the sequential AMR discussion, various levels of optimizations were introduced and the pros and cons of each were analysed in isolation and with each other.

Finally, two versions of multi-threaded AMR were used for experiments :

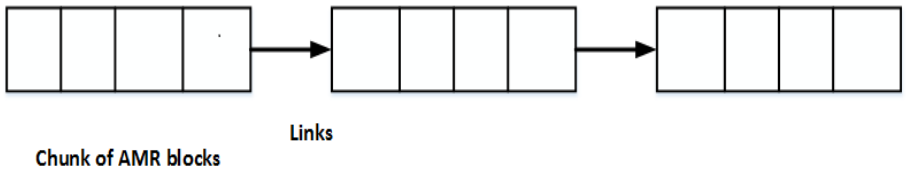


Figure 5.7: Concurrent Linked list of AMR blocks

1. Shared block array : Memory for AMR blocks local to an MPI rank are allocated in chunks and stored in a single concurrent linked list 5.7. Blocks are numbered starting from 0 to $N-1$ on each process. These blocks are assigned to threads in the order of thread ids. New blocks may be added during refinement and existing blocks may be deleted during coarsening. Frequent changes in the number of blocks leads to load imbalance which is removed by reassigning blocks to threads. Load balance across MPI ranks is handled by the amortization routine which triggers reassignment automatically. All accesses to the linked list of blocks are through atomic load/store instructions. In this version of AMR, only the leader thread (thread 0) can add and delete block chunks. Mapping of blocks to threads is unique, any thread can traverse the linked list in parallel, access the blocks it owns and modify its data. This works out fine, because the simulation progresses in stages. Links are added or deleted from the list only after refinement. Thread 0 counts the number of new blocks required and attaches them to the list. This is a constant time operation. After blocks are allocated, threads copy their data in parallel to the list. They synchronize after refinement and update the dictionary.

2. Thread local arrays of blocks : In this version, instead of a single linked-list, we maintain a vector of linked-lists, one list for each thread. Like in the previous implementation, blocks are allocated and deleted in chunks. The addition and deletion of blocks after refinement are contention-free operations local to each thread.

5.3.2 Multi-threaded Algorithms for AMR

We have tried to make the algorithms as contention free as possible by partitioning blocks between threads, where each thread owns a set of blocks. Auxiliary vectors were used as scratchpad to remember state information during the execution of these algorithms.

5.3.2.1 Multi-threaded Refinement

We have tried to make the algorithms as contention free as possible by partitioning blocks between threads, where each thread owns a set of blocks. Auxiliary vectors were used as scratchpad to remember state information during the execution of these algorithms.

5.3.2.2 Multi-threaded Refinement

The pseudo-codes for modified refinement algorithms are provided in algorithm 5.6 and 5.7. *Thread_prefix* is a function that computes parallel prefix over the values provided by each thread. Each block has integer fields it shares with each of its neighbors for communicating with them, called *nbr_ref*. These fields store refinement states of block neighbors as $\{-1, 0, 1\}$. This communication is contention free since fields are updated by neighbors and read by owning threads at different points in time. These writes and reads are ordered and separated in time by a fence and a thread barrier. The vector *refine_list* marks all blocks that are ready for a transition in the current iteration. If a block is marked for transition ($nbr\_ref = 1$), it enters the refinement routine and conveys this information to siblings and neighbors. If siblings or neighbors are local, these exchanges occur through reads and writes. If they are remote, a message is created for each non-local sibling and neighbor and aggregated like in the previous version. Message aggregation is done in parallel by threads. After traversing through blocks once, threads block and enter a round of update, called *State_transition*, where they read *nbr_ref* fields and update their own states. The *State_transition* routine is invoked twice in this implementation. The first execution of this routine is to incorporate local neighbor updates. The second execution is after exchanging remote neighbor updates. Whenever a block transitions to *stay* or *refine*, its location in the *refine_list* is set. Otherwise, the corresponding *refine_list* value is reset. This updated *refine_list* is used

Algorithm 5.6 Multi-threaded Consensus Algorithm

```
1:  procedure M_CONSENSUS
2:      THREAD_PREFIX(prefix, mylocal)
3:      while ¬Terminate do
4:          while ¬Local_Terminate do
5:              for all n ∈ range[1 : mylocal] do
6:                  if refine_list[prefix + n] then
7:                      refine_list[prefix + n] ← 0
8:                      nbrs = blocks[n].nbrlist()
9:                      sibs = blocks[n].slist()
10:                     for all nbr ∈ nbrs do
11:                         if nbr.is_local() then
12:                             LOCAL_UPDATE_NBR_REF(n, nbr)
13:                         else
14:                             aggregate_msg(nbr)
15:                         end if
16:                     end for
17:                     for all sib ∈ sibs do
18:                         if sib.is_local() then
19:                             LOCAL_UPDATE_SIB_REF(n, sib)
20:                         else
21:                             aggregate_msg(sib)
22:                         end if
23:                     end for
24:                 end if
25:             end for
26:             STATE_TRANSITION
27:         end while
28:         MPI_Neighbor_alltoallv()
29:         STATE_TRANSITION
30:         MPI_alltoall(quiescence)
31:     end while
32: end procedure
```

in the next iteration.

Algorithm 5.7 local_state_update

```
1:  procedure STATE_TRANSITION
2:      THREAD_PREFIX(prefix, mylocal)
3:      for all n ∈ range[1 : mylocal] do
4:          sibs = blocks[n].slist()
5:          nbrs = blocks[n].nbrlist()
6:          for all do sib ∈ sibs
7:              if then CANNOT_COARSEN(n, sib, state)
8:                  blocks[n].refine ← state
9:                  refine_list[prefix + n] ← 1
10:             end if
11:         end for
12:         for all do nbr ∈ nbrs
13:             if then REFINE_OR_STAY_BLOCK(n, nbr, state)
14:                 blocks[n].refine ← state
15:                 refine_list[prefix + n] ← 1
16:             end if
17:         end for
18:     end for
19: end procedure
```

In our current implementation, the *refine_list* is implemented as a vector, unlike the previous worklist implementation. This made the implementation contention-free since array locations are distinct. A multi-producer, multi-consumer concurrent queue can become a bottleneck in the refinement algorithm if used as worklist. There are two atomic variables that are used as entry gates to the refinement algorithm - *terminate* and *local_terminate*. For an MPI rank, if all entries in its *refine_list* are reset, the *local_terminate* variable tests *true*. Remote data is exchanged using an *MPI_Alltoallv* call after this phase. The *refine_list* is examined after the second *State_transition* call. If all entries in the *refine_list* are reset, a $token$ variable is assigned 0, else it is assigned 1, indicating membership in the next refinement iteration. All $token$ variables are combined using $MPI\_allreduce$ and the $MAXIMUM$ operation. If the result is 0, then *terminate* turns *true*. In this version, although packing and unpacking of messages is done by threads in parallel, MPI calls are invoked only by thread 0. The remaining threads block until the communication is over. The first set of testcases cover shared memory performance on a single node, for various memory configurations. Intel KNL nodes have two types of memory - DDR and MCDRAM. DDR is low latency and low bandwidth memory, while MCDRAM has high latency and high bandwidth. A table

of latencies for both memory units on KNL can be found here [80].

A brief discussion of the memory modes [80] used in our experiments are provided here:

1. Cache Quadrant : The fast memory (MCDRAM) is used as shared L3 cache, with total size 16GB. The tiles on KNL are divided into 4 groups, hence it is also called as quadrant mode. The cache is direct-mapped and address look-up is done in a distributed manner in quadrants. The percentage of fast memory used as cache is configurable. We used the configuration where the entire MCDRAM was used as cache. This configuration gave good performance for most of the testcases discussed in this thesis. Unless otherwise mentioned, this is the default configuration in which we have used KNL nodes.

2. Flat Quadrant : This configuration has one NUMA region and two NUMA nodes - MCDRAM and DDR. Memory allocation can be done selectively in each of these memories depending on the size of data, and its frequency of access. For the AMR testcases, the largest data structures are the blocks, which are frequently read and written by every routine in every timestep. They were allocated in fast memory and auxiliary arrays were allocated in DDR. When the fast memory was not enough, allocation was shifted to DDR.

3. SNC4 : This configuration has 4 NUMA regions and 8 NUMA nodes - 4 DDR nodes and 4 MCDRAM nodes.

The testcase used for on-node performance measurements had an initial mesh of 2048 blocks, blocksize is $14X14X14$ and each point had 10 variables. The average number of blocks per thread is shown in the tables. The blocks are arranged in Morton order after load balancing. We did not optimize load balancing for these experiments. The graphs below show strong scaling for increasing number of threads per node. All measurements were taken on Intel KNL nodes for a total of 1000 iterations. The reported times are aggregated over 1000 iterations.

The tables 5.1 and 5.2 and the graph 5.8 measure the performance of the refinement algorithm and stencil phase. New blocks created during refinement are added to the block array in random order. At the end of the refinement stage, SFC order of blocks is violated. Load balancing was invoked after every refinement stage to re-order the blocks. This included sorting blocks in memory. The refinement frequency for this testcase was 2, and load balancing was performed 500 times out of 1000 iterations. We

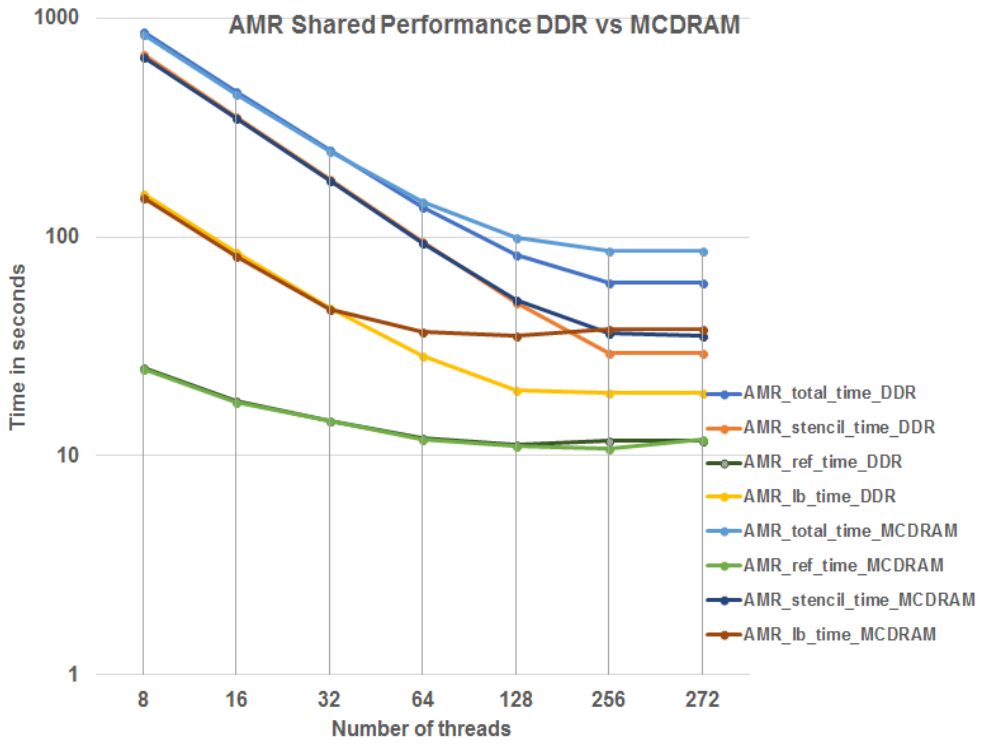


Figure 5.8: AMR Performance on single KNL node for DDR and MCDRAM, with shared block array

#threads	avg_blocks	Ref.Time	Stencil.time	Lb.Time	Total.Time
8	344	25.2134	680.301	157.067	866.692
16	172	17.7512	353.855	85.5643	459.657
32	84	14.5228	184.724	47.3972	248.213
64	43	12.0266	95.6273	28.8518	137.768
128	22	11.2854	50.1697	20.0627	82.6781
256	12	11.7977	29.5161	19.5418	62.1841
272	11	11.7816	29.484	19.4494	62.1486

Table 5.1: AMR shared memory performance : flat quadrant DDR

#threads	avg_blocks	Ref.Time	Stencil.time	Lb.Time	Total.Time
8	344	24.8698	663.91	152.104	844.941
16	172	17.6279	349.369	81.5142	451.076
32	84	14.3761	182.096	47.0013	245.022
64	43	11.9677	94.5218	37.0457	144.797
128	22	11.1479	51.9063	35.6126	99.8848
256	12	10.7662	36.3458	38.1115	86.5989
272	11	11.8714	35.2775	38.0808	86.8349

Table 5.2: AMR shared memory performance : flat quadrant MCDRAM

changed this implementation later to lower load balancing cost. The optimizations are explained in detail below.

#threads	avg_blocks	Ref.Time	Stencil.time	Lb.Time	Total.Time
8	473	11.7909	584.12	3.33322	604.568
16	237	7.5639	321.268	2.53552	334.374
32	119	5.49214	268.981	2.15876	278.503
64	60	3.94208	185.125	1.78557	192.024
128	30	3.52678	159.139	1.7984	165.491
256	15	4.29064	131.326	2.00158	139.047
272	14	4.78033	124.817	2.65953	133.929

Table 5.3: AMR shared memory performance : SNC4 DDR

#threads	avg_blocks	Ref.Time	Stencil.time	Lb.Time	Total.Time
8	473	11.8432	588.002	3.32723	608.495
16	237	7.55572	313.182	2.44411	326.179
32	119	5.40252	169.19	1.74717	178.214
64	60	3.95622	94.0587	1.26582	100.464
128	30	3.71264	66.9747	1.14642	72.8951
256	15	4.56711	46.998	1.4305	54.3404
272	14	5.06215	45.9796	1.74826	54.4818

Table 5.4: AMR shared memory performance : SNC4 MCDRAM

The second testcase is initialized to the same size, 2048 blocks. Coarsening was turned off for this test. All measurements were performed in the SNC mode with numa-aware memory allocation. The table 5.3 is SNC mode with DDR memory, which uses 4 of the 8 numa nodes. All block arrays were allocated on two of the numa nodes, without considering the cores to which threads were pinned. No memory was allocated on MCDRAM. The second table has the same testcase with memory allocated on 4 MCDRAM numa nodes. The difference in performance is attributed to a couple of factors :

1. Hierarchical load balancing, with Morton order preserved in each thread local array
2. Cheaper load balancing, since blocks are already in sorted order across threads.
3. Careful memory allocation, arrays were allocated on the numa node closest to the quadrant which contained the thread.

A graph containing the comparison is plotted in 5.9;

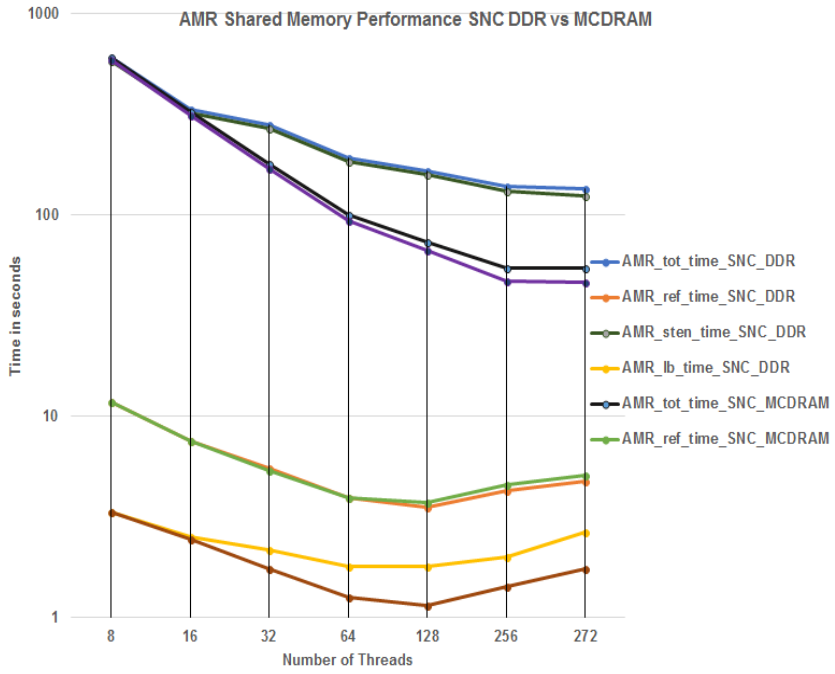


Figure 5.9: AMR Performance on single KNL node for DDR and MCDRAM with 8 numa nodes and thread local arrays

5.3.3 Stencil Computation

The stencil computation phase has full overlap of computation and communication. Threads accumulate messages for remote neighbors and pack them in the send buffer in parallel. If running with multiple MPI ranks, thread 0 on every MPI rank starts a non-blocking neighbor collective (alltoallv) to exchange these messages. While waiting for the nearest neighbor communication to complete, threads work on computing the stencil at points within the volume of each block. They also exchange boundaries with local neighbors. Each block has one extra buffer in every dimension to store in-coming halo messages from neigbors. When the neighbor collective completes, the receive buffer is unpacked in parallel and halo messages are stored in their intended blocks. After all halos are received, new boundaries are computed.

5.3.4 Load Balance

Load balancing is done in stages. All current versions of multi-threaded AMR maintain blocks in sorted order at all times. Let R_i indicate the rank of an MPI process with

Algorithm 5.8 Multi-threaded_SFC_slicing

```
1: procedure MULTI-THREADED_SFC_SLICING
2:     PARALLEL_SFC_SLICING
3:     for all do n ∈ range[1 : myload]
4:         if then n.assign ≠ mype
5:             PACK_IN_BUFFER(n)
6:         end if
7:     end for
8:     MPI_Neighbor_alltoallv()
9:     for do n ∈ range[1 : recvblocks]
10:        if then recv_proc(n) < mype
11:            UNPACK_BLOCK(n, 0)
12:        else UNPACK_BLOCK(n, num_threads − 1)
13:        end if
14:    end for
15: end procedure
```

index i. SFC keys on all MPI processes are stored in sorted order. Between processes, they are stored in increasing order, according to process ranks. For any two MPI ranks R_i and R_j, all keys on R_i are strictly less than keys on R_j where $R_i < R_j$. Refinement and coarsening do not violate this condition. Suppose two blocks b_i and b_j are refined, and $key(b_i) < key(b_j)$. Let $b_{i1}, ..., b_{i8}$ be the keys of b_i's subcells and $b_{j1}, ..., b_{j8}$ be the keys of b_j's subcells. Then, all keys belonging to the set $\{b_{i1}, ..., b_{i8}\} < \{b_{j1}, ..., b_{j8}\}$.

When new blocks are added, existing blocks are shifted to reserve space and they are inserted into their correct position.

In the version with shared arrays, thread 0 computes the new location of blocks, creates the communication graph, along with source and destination processes. If running with multiple MPI ranks, blocks are packed into the send buffer by owning threads in parallel and thread0 on all ranks executes the sparse collective for data transfer. The receive buffer is unpacked in parallel by threads and new blocks inserted into the locations they belong to.

We have called this incremental load balancing. The pseudo-code for the slicing algorithm, that does incremental load balancing is provided below :

5.3.5 Distributed AMR

This is the hybrid version of AMR with one MPI rank per node and multiple threads per node. The initial partitions and orders are generated by our partitioning framework, by first constructing a distributed kd-tree, followed by parallel SFC traversal. We have used Morton order for the results in this section. The initial mesh had $64X64X32$ blocks, where each block had dimensions $14X14X14$ and 10 observations per point. The total initial mesh size is 359661568 points with 10 doubles per point. The domain had an object in its lower left corner, which caused the mesh to refine every 2 timesteps. The maximum refinement level is 5. The number of processes ranges from $8 - 256$ and the number of threads from $64 - 256$. The number of cores ranges from $[512 - 16384]$ and the total number of threads from $[512 - 65536]$. Total execution time is the sum of $Ref + Data$ and $stencil$. All reported times are measured according to the equations in 2.2.

Amortized load balancing was enabled in the hybrid version to detect variations in load. Load balancing in the hybrid version was done in three stages :

1. On-node load balancing : This is the re-assignment of blocks within threads on a node. Its a simple parallel prefix operation, relatively cheap, if blocks are already sorted.

2. Incremental adjustments : Processes compute a parallel-prefix with their local loads and determine the ranks of all blocks they own. If ranks are out of range, then those blocks are packed and sent to their home processes. But for small variations in load, this is an incremental exchange where processes exchange blocks with their neighbors.

3. Full re-assignment : For large variations in load and/or significant changes in the domain, it may be beneficial to re-compute the kd-tree and traverse blocks to generate a new permutation.

For the test case here, full re-assignment was done once when the simulation was initialized. On-node load balancing was done at every step. Incremental changes were detected by the amortization equation. This is included in the total execution time.

The measured values are tabulated in table 5.5. The implementation seems to scale

#ranks	th	avgblocks	Ref.Time	Ref+copy	Sten.time	Total.Time
8	64	17088	2.35669	112.519	38.422	189.37
8	128	17088	2.6576	140.442	26.6923	211.702
8	256	17088	4.62349	267.926	27.4765	377.958
16	64	8544	1.08933	57.034	16.8705	93.2348
16	128	8544	1.17652	71.584	12.7828	107.255
16	256	8544	2.08195	134.572	13.7747	190.748
32	64	4272	0.554039	29.7559	8.58164	48.6074
32	128	4272	0.664393	38.9083	6.94482	58.8311
32	256	4272	1.1517	69.4193	7.14011	100.579
64	64	2136	0.338868	17.1698	4.60219	28.0335
64	128	2136	0.486048	22.6756	3.78187	35.1796
64	256	2136	0.855437	37.5925	4.02513	54.5667
100	64	1367	0.277801	12.8137	3.20701	20.6649
100	128	1367	0.446624	17.2612	2.53417	26.1836
100	256	1367	0.780824	27.6433	2.61986	39.5421
128	64	1068	0.259182	10.7914	2.61336	17.3003
128	128	1068	0.411999	14.245	2.18787	21.4391
128	256	1068	0.751798	23.2749	2.28296	33.269
150	64	911	0.24468	9.87265	2.38948	15.9123
150	128	911	0.485077	14.5655	2.00325	21.5884
150	256	911	0.759823	21.0147	2.01838	30.2441
200	64	683	0.228815	8.5146	1.83267	13.4199
200	128	683	0.437202	12.1182	1.5976	17.8738
200	256	683	0.737671	18.8441	1.65367	27.3698
256	64	534	0.229304	7.45703	1.55439	11.6733
256	128	534	0.475408	10.7538	1.44482	15.9098
256	256	534	0.724172	15.6654	1.51874	22.6364

Table 5.5: AMR Hybrid Shared

well, with increasing number of MPI ranks, but not with increasing number of threads. The graphs are plotted in figure 5.10. We have used the shared memory version with a single linked-list here. This version didnot scale well. This performance is expected. $Ref + Data$ includes the time for refinement/coarsening as well as re-ordering blocks according to their keys, i.e insert new blocks in key order and remove holes in the arrays where blocks are deleted.

#MPI_ranks	threads	Ref.Time	Stencil.time	Total.Time
8	64	4.76182	32.5261	43.0979
8	128	5.10738	21.5452	33.1129
8	256	8.51228	20.9566	38.7395
16	64	4.11538	17.2043	25.8236
16	128	4.80708	12.663	22.8252
16	256	8.04316	12.3869	28.6362
32	64	3.91559	9.69997	17.8178
32	128	4.71262	7.37938	16.7505
32	256	7.78562	7.89839	22.6205
64	64	4.21512	6.14988	13.3933
64	128	4.66342	4.7967	12.9276
64	256	7.42973	5.59758	18.3588
100	64	3.97731	4.49775	10.9322
100	128	4.55437	3.77888	11.1412
100	256	4.9076	10.7202	19.7556
128	64	3.75452	3.94103	9.8955
128	128	4.62398	3.40179	10.9427
128	256	4.86113	10.2732	19.0711
150	64	3.92705	3.59772	9.64096
150	128	2.87188	7.44273	12.7336
150	256	3.4613	5.83619	12.8414
200	64	3.86591	3.10235	9.0912
200	128	3.50961	2.28771	7.80373
200	256	3.3612	5.57249	12.0393
256	64	3.9594	2.97883	9.01548
256	128	2.90627	6.92563	12.1595
256	256	3.4317	5.42846	12.0124

Table 5.6: AMR Hybrid Thread Local Arrays

The improved version with thread local arrays for blocks had better performance due to higher parallelism and lower synchronization. All algorithms in the multi-threaded version benefit from reduced synchronization. The total refinement time is considerably lower for this version, addition and deletion of blocks is done in parallel by threads to their local arrays. In the shared array, some of these operations are sequential. The observations with reduced execution times for thread local arrays are

tabulated in table 5.6. They are also plotted in the graph 5.11. This version scales well until 128 threads. The performance dropped at 256 threads. A comparison between both implementations is provided in the graph 5.12.

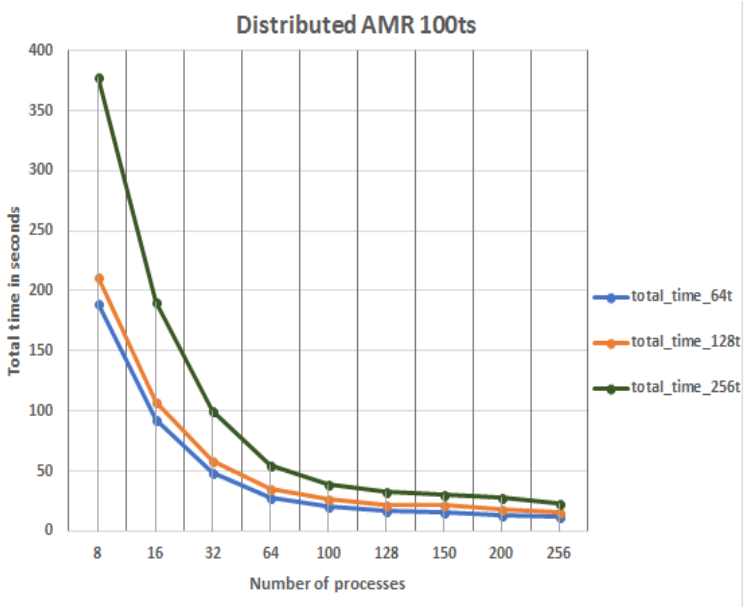


Figure 5.10: AMR Performance on multiple KNL nodes, with shared array implementation

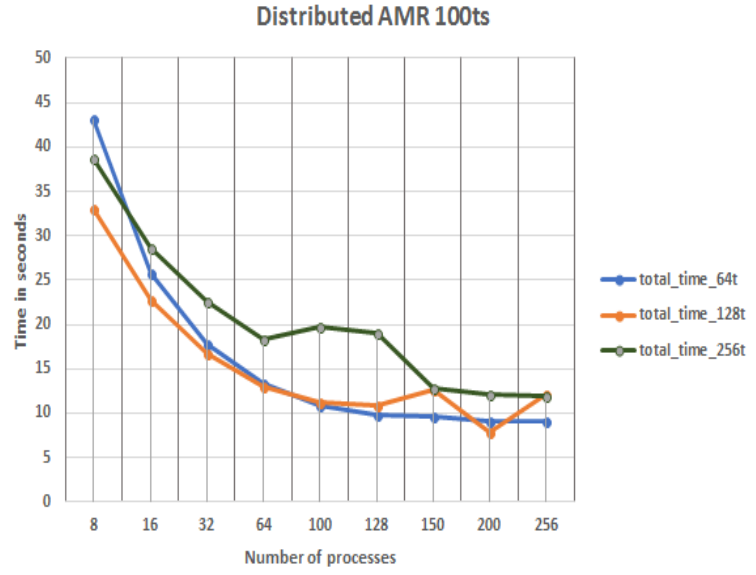


Figure 5.11: AMR Performance on multiple KNL nodes, with thread local arrays implementation

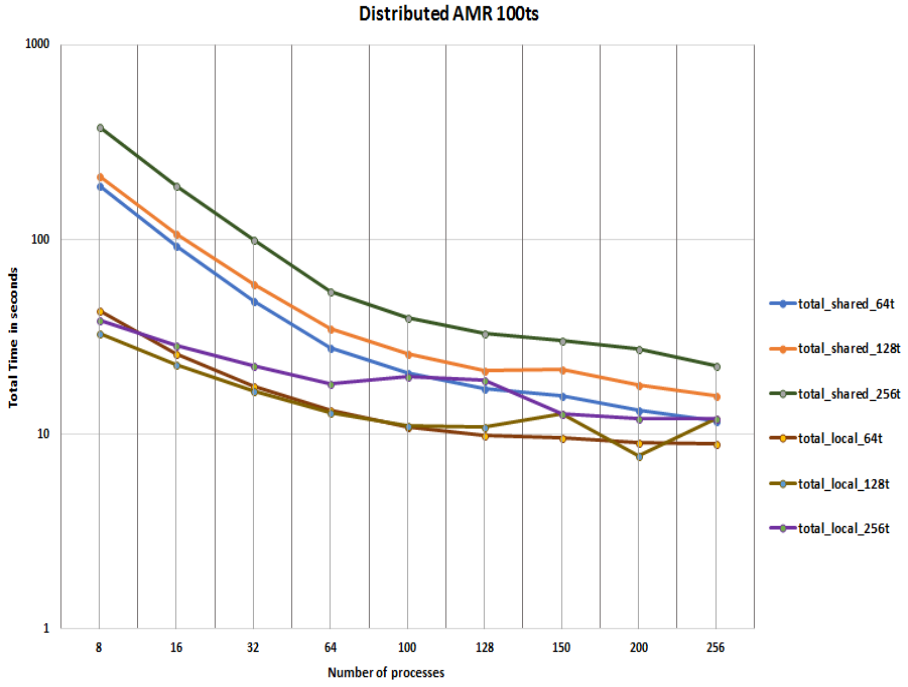


Figure 5.12: Comparison of distributed AMR implementation (shared vs with thread local arrays)

CHAPTER 6: CONCLUSION

We have developed a distributed multi-threaded data partitioner that can be used to partition meshes, points with co-ordinates. The partitioner is compared to multi-level graph partitioners like Metis and Scotch. We have also evaluated the performance of the partitioner for various dimensions, data sizes, number of processes and threads. Strong scaling results are presented for different splitters and space-filling curves. We have also evaluated the management of dynamic data, using an iterative benchmark. Incremental modifications to partitions are evaluated for different dimensions and problem sizes. The partitioner is included in a benchmark for adaptive mesh refinement. We have presented both strong and weak scaling results for this AMR benchmark, with options for incremental load balancing within a process and across processes. Besides incremental adjustments, there are times when a full re-partition is desired. This depends on the application and the total number of timesteps in the simulation. An example would be the simulation of a moving object which changes its location in the mesh. The dataset for such a simulation will have clusters around the center of gravity of the object, which shifts when the object moves. For such testcases, the hierarchical tree decomposition can become unbalanced, with long paths in the tree which make its maximum depth $O(n)$, instead of $O(logn)$. A space-filling curve defined on this tree looses its shape and it is highly likely that partitions have higher surface area for the same volume. To accomodate such cases, we have options to invoke a full repartition and re-assignment at certain timesteps during a large simulation. The other option is to allow incremental adjustments to the kd-tree that re-balance it whenever nodes are adjusted. One can choose any balance criteria and ensure it is not violated, e.g the maximum depth of any two sibling subtrees should not differ by more than one. Re-balancing tree nodes is an option to track midpoints and medians of a dynamic data distribution. We have not implemented this yet, it is a possible option for future work. Another direction for future work is to include adjacency edges in splitter computation. This may make a difference to the quality of unstructured mesh partitions. There are two possible ways to partition general graphs using SFCs - either partition the adjacency matrix or partition the embedded mesh. Embedding is a pre-processing step required before partitioning general graphs, if this direction seems feasible. It has been adopted for partitioning graphs from social networks which tend to have very high vertex degrees. Finding good embedding functions that satisfy constraints and relationships in the original graph is a hard problem that requires further investigation.

Statistical evaluation of pointsets to reduce the number of dimensions is a possible optimization that can greatly reduce the problem size and improve execution time. A combination of these two approaches need to be applied to general graphs before they are partitioned - find a minimal set of relationships that need to be maintained, followed by embedding. These relationships are also susceptible to change depending on the problem or query set. Therefore, we are in a domain with dynamic graphs with changing relationships (edges) that need efficient embeddings and partitioning. There is some recent work in this direction [118].

REFERENCES

[1] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. ACM, 1997. [Online]. Available: http://doi.acm.org/10.1145/263580.263662 pp. 340–347.

[2] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," *SIGPLAN Not.*, vol. 26, no. 4, pp. 63–74, Apr. 1991. [Online]. Available: http://doi.acm.org/10.1145/106973.106981

[3] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC'09. Springer-Verlag, 2010, pp. 50–64.

[4] K. Schloegel, G. Karypis, and V. Kumar, *Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes*. IEEE Press, May 2001, vol. 12, no. 5. [Online]. Available: http://dx.doi.org/10.1109/71.926167

[5] C. Chevalier and F. Pellegrini, "Pt-scotch: A tool for efficient parallel graph ordering," *Parallel Comput.*, vol. 34, no. 6-8, pp. 318–331, July 2008. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2007.12.001

[6] W. Wang, J. Yang, and R. Muntz, *Information Organization and Databases*, K. Tanaka, S. Ghandeharizadeh, and Y. Kambayashi, Eds. Kluwer Academic Publishers, 2000. [Online]. Available: http://dl.acm.org/citation.cfm?id=571220.571247

[7] H. Samet, *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[8] D. Miranker, W. Xu, and R. Mao, "Mobios: a metric-space dbms to support biological discovery," in *15th International Conference on Scientific and Statistical Database Management, 2003.*, July 2003, pp. 241–244.

[9] R. Agrawal, C. Faloutsos, and A. N. Swami, "Efficient similarity search in sequence databases," in *Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms*, ser. FODO '93. Springer-Verlag, 1993. [Online]. Available: http://dl.acm.org/citation.cfm?id=645415.652239 pp. 69–84.

[10] V. S. Subrahmanian, *Principles of Multimedia Database Systems*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[11] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the positions of continuously moving objects," *SIGMOD Rec.*, vol. 29, no. 2, pp. 331–342, May 2000. [Online]. Available: http://doi.acm.org/10.1145/335191.335427

[12] V. Pascucci and R. J. Frank, "Global static indexing for real-time exploration of very large regular grids," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, ser. SC '01. ACM, 2001. [Online]. Available: http://doi.acm.org/10.1145/582034.582036 pp. 2–2.

[13] H. Sagan, "A three-dimensional hilbert curve," *International Journal of Mathematical Education in Science and Technology*, vol. 24, no. 4, pp. 541–545, 1993.

[14] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[15] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, Sep. 1982. [Online]. Available: http://doi.acm.org/10.1145/356887.356892

[16] 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.*, vol. 74, no. 12, pp. 3217–3227, Dec. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2014.07.001

[17] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, Dec. 1986.

[18] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," ser. DAC '99. ACM, 1999. [Online]. Available: http://doi.acm.org/10.1145/309847.309954 pp. 343–348.

[19] D. Delling, A. V. Goldberg, I. P. Razenshteyn, and R. F. F. Werneck, "Exact combinatorial branch-and-bound for graph bisection," in *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX 2012, The Westin Miyako, Kyoto, Japan, January 16, 2012*, 2012. [Online]. Available: https://doi.org/10.1137/1.9781611972924.3 pp. 30–44.

[20] W. W. Hager, D. T. Phan, and H. Zhang, "An exact algorithm for graph partitioning," *Math. Program.*, vol. 137, no. 1-2, pp. 531–556, 2013. [Online]. Available: https://doi.org/10.1007/s10107-011-0503-x

[21] T. Leighton and S. Rao, "Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms," *J. ACM*, vol. 46, no. 6, pp. 787–832, Nov. 1999. [Online]. Available: http://doi.acm.org/10.1145/331524.331526

[22] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck, *Graph Partitioning with Natural Cuts*, 2011.

[23] A. H. Gebremedhin, A. Tarafdar, F. Manne, and A. Pothen, "New acyclic and star coloring algorithms with application to computing hessians," *SIAM J. Sci. Comput.*, vol. 29, no. 3, pp. 1042–1072, May 2007. [Online]. Available: http://dx.doi.org/10.1137/050639879

[24] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, *ColPack: Software for Graph Coloring and Related Problems in Scientific Computing*. New York, NY, USA: ACM, Oct. 2013, vol. 40, no. 1. [Online]. Available: http://doi.acm.org/10.1145/2513109.2513110

[25] W. E. Donath and A. J. Hoffman, "Lower bounds for the partitioning of graphs," *IBM J. Res. Dev.*, vol. 17, no. 5, pp. 420–425, Sep. 1973. [Online]. Available: http://dx.doi.org/10.1147/rd.175.0420

[26] M. Fiedler, "Algebraic connectivity of graphs," *Czechoslovak Mathematical Journal*, vol. 23, no. 2, pp. 298–305, 1973. [Online]. Available: http://dml.cz/dmlcz/101168

[27] T. M. S. U. Maxim Naumov (NVIDIA), "Parallel spectral graph partitioning," NVIDIA, Tech. Rep., 2016.

[28] S. B.W.Kernighan, "An efficient heuristic procedure for partitioning graphs," Tech. Rep., 1970.

[29] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proceedings of the 19th Design Automation Conference*, ser. DAC '82. Piscataway, NJ, USA: IEEE Press, 1982. [Online]. Available: http://dl.acm.org/citation.cfm?id=800263.809204 pp. 175–181.

[30] J. W. Marks, "Graph partitioning system," Patent, 1998.

[31] J. R. Gilbert, G. L. Miller, and S.-H. Teng, "Geometric mesh partitioning: Implementation and experiments," *SIAM J. Sci. Comput.*, vol. 19, no. 6, pp. 2091–2110, 1998. [Online]. Available: http://dx.doi.org/10.1137/S1064827594275339

[32] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '12. ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2339530.2339722 pp. 1222–1230.

[33] J. Leskovec and R. Sosič, *SNAP: A General-Purpose Network Analysis and Graph-Mining Library*. New York, NY, USA: ACM, July 2016, vol. 8, no. 1. [Online]. Available: http://doi.acm.org/10.1145/2898361

[34] S. Sakr, F. M. Orakzai, I. Abdelaziz, and Z. Khayyat, *Large-Scale Graph Processing Using Apache Giraph*, 1st ed. Springer Publishing Company, Incorporated, 2017.

[35] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USENIX Association, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685096 pp. 599–613.

[36] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273005

[37] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522738 pp. 439–455.

[38] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: https://doi.org/10.14778/2212351.2212354

[39] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465369 pp. 169–182.

[40] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184 pp. 135–146.

[41] R. A. Johnson and D. W. Wichern, Eds., *Applied Multivariate Statistical Analysis*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[42] R. J. Lipton, D. J. Rose, and R. E. Tarjan, "Generalized nested dissection," *SIAM Journal on Numerical Analysis*, vol. 16, no. 2, pp. 346–358, 1979. [Online]. Available: http://www.jstor.org/stable/2156840

[43] M. T. Heath and P. Raghavan, "A cartesian parallel nested dissection algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 16, no. 1, pp. 235–253, 1995.

[44] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69. ACM, 1969. [Online]. Available: http://doi.acm.org/10.1145/800195.805928 pp. 157–172.

[45] S.-H. Teng, "Fast nested dissection for finite element meshes," *SIAM Journal on Matrix Analysis and Applications*, vol. 18, no. 3, pp. 552–565, 1997.

[46] W. Aiello, F. Chung, and L. Lu, "A random graph model for power law graphs," *Experimental Math*, vol. 10, pp. 53–66, 2000.

[47] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998. [Online]. Available: http://dx.doi.org/10.1137/S1064827595287997

[48] C. Chevalier and F. Pellegrini, "Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework," in *Euro-Par 2006 Parallel Processing: 12th International Euro-Par Conference, Dresden, Germany, August 28 –September 1, 2006. Proceedings*. Springer Berlin Heidelberg, 2006, pp. 243–252.

[49] F. Pellegrini, "A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries," in *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par'07. Springer-Verlag, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=2391541.2391566 pp. 195–204.

[50] M. C. Walshaw and K.McManus, "Multiphase mesh partitioning," *Applied Mathematical Modelling*, pp. 123–140, 2000.

[51] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen, "A repartitioning hypergraph model for dynamic load balancing," *J. Parallel Distrib. Comput.*, vol. 69, no. 8, pp. 711–724, Aug. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2009.04.011

[52] B. Hendrickson, "Graph partitioning and sequencing software, version 00," Tech. Rep., 9 1995.

[53] P. Sanders and C. Schulz, "Think Locally, Act Globally: Highly Balanced Graph Partitioning," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, ser. LNCS, vol. 7933. Springer, 2013, pp. 164–175.

[54] B. Monien and S. Schamberger, "Graph partitioning with the party library: helpful-sets in practice," in *16th Symposium on Computer Architecture and High Performance Computing*, 2004, pp. 198–205.

[55] Ü. V. Çatalyürek and C. Aykanat, "Patoh (partitioning tool for hypergraphs)," in *Encyclopedia of Parallel Computing*, 2011, pp. 1479–1487.

[56] H. Meyerhenke, B. Monien, and T. Sauerwald, "A new diffusion-based multilevel algorithm for computing graph partitions," *J. Parallel Distrib. Comput.*, vol. 69, no. 9, pp. 750–761, Sep. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2009.04.005

[57] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 1055–1064.

[58] Q. Du, V. Faber, and M. Gunzburger, "Centroidal voronoi tessellations: Applications and algorithms," *SIAM Rev.*, vol. 41, no. 4, pp. 637–676, Dec. 1999. [Online]. Available: http://dx.doi.org/10.1137/S0036144599352836

[59] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008.

[60] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[61] J. Kim, S.-G. Kim, and B. Nam, "Parallel multi-dimensional range query processing with r-trees on gpu," *J. Parallel Distrib. Comput.*, vol. 73, no. 8, pp. 1195–1207, Aug. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2013.03.015

[62] I. Al-Furaih, S. Aluru, S. Goil, and S. Ranka, "Parallel construction of multidimensional binary search trees," *IEEE Trans. Parallel Distrib. Syst.*, vol. 11, no. 2, pp. 136–148, Feb. 2000. [Online]. Available: http://dx.doi.org/10.1109/71.841750

[63] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and S.-H. Teng, "Constructing trees in parallel," in *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '89. ACM, 1989. [Online]. Available: http://doi.acm.org/10.1145/72935.72980 pp. 421–431.

[64] K. Hinrichs, "Implementation of the grid file: Design concepts and experience," *BIT*, vol. 25, no. 4, pp. 569–592, Dec. 1985. [Online]. Available: http://dx.doi.org/10.1007/BF01936137

[65] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975.

[66] T. J. Tauges and R. Jain, "Creating geometry and mesh models for nuclear reactor core geometries using a lattice hierarchy-based approach," *Engineering with Computers*, vol. 28, pp. 319–329, 2012.

[67] A. Appel, "Some techniques for shading machine renderings of solids," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring). ACM, 1968. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468082 pp. 37–45.

[68] M. C. Lin, "Efficient collision detection for animation and robotics," Ph.D. dissertation, 1993, aAI9430587.

[69] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," *ACM Trans. Graph.*, vol. 27, no. 5, pp. 126:1–126:11, Dec. 2008. [Online]. Available: http://doi.acm.org/10.1145/1409060.1409079

[70] M. Shevtsov, A. Soupikov, and A. Kapustin, "Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes." *Comput. Graph. Forum*, vol. 26, no. 3, pp. 395–404, 2007.

[71] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *J. Comput. Syst. Sci.*, vol. 7, no. 4, pp. 448–461, Aug. 1973. [Online]. Available: http://dx.doi.org/10.1016/S0022-0000(73)80033-9

[72] *Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7*, Std., Sept 2016.

[73] B. Stroustrup, *The C++ Programming Language*, 4th ed. Addison-Wesley Professional, 2013.

[74] 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.

[75] R. C. Murphy, K. B. Wheeler, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *2008 IEEE International Parallel and Distributed Processing Symposium(IPDPS)*, vol. 00, 04 2008. [Online]. Available: doi.ieeecomputersociety.org/10.1109/IPDPS.2008.4536359 pp. 1–8.

[76] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[77] M. E. Conway, "A multiprocessor system design," in *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, ser. AFIPS '63 (Fall). ACM, 1963. [Online]. Available: http://doi.acm.org/10.1145/1463822.1463838 pp. 139–146.

[78] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[79] M. Snir, "Depth-size trade-offs for parallel prefix computation," *J. Algorithms*, vol. 7, no. 2, pp. 185–201, June 1986. [Online]. Available: http://dx.doi.org/10.1016/0196-6774(86)90003-9

[80] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2Nd Edition*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[81] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," May 2013. [Online]. Available: http://www.openmp.org/specifications

[82] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, ser. FOCS '99. IEEE Computer Society, 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=795665.796479 pp. 285–.

[83] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998. [Online]. Available: http://doi.acm.org/10.1145/272991.272995

[84] "Texas advanced computing center (tacc)." [Online]. Available: http://www.tacc.utexas.edu

[85] M. P. Forum, *MPI: A Message-Passing Interface Standard*, Std., 1994.

[86] J. Bruck, C.-T. Ho, E. Upfal, S. Kipnis, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 11, pp. 1143–1156, Nov. 1997. [Online]. Available: https://doi.org/10.1109/71.642949

[87] J. Kleinberg and E. Tardos, *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[88] W. Schroeder, K. M. Martin, and W. E. Lorensen, "The visualization toolkit (2nd ed.): An object-oriented approach to 3d graphics," Upper Saddle River, NJ, USA, Tech. Rep., 1998.

[89] T. Ringler, L. Ju, and M. Gunzburger, "A multiresolution method for climate system modeling: application of spherical centroidal voronoi tessellations," *Ocean Dynamics*, vol. 58, no. 5-6, 11 2008.

[90] J. Snyder, "Map projections – a working manual," Tech. Rep., 01 1987.

[91] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, "The ibm blue gene/q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1109/MM.2011.108

[92] "Argonne national laboratory(anl)." [Online]. Available: www.anl.gov

[93] M. S. Warren and J. K. Salmon, "A parallel hashed oct-tree n-body algorithm," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993. [Online]. Available: http://doi.acm.org/10.1145/169627.169640 pp. 12–21.

[94] H. Si, "Tetgen, a delaunay-based quality tetrahedral mesh generator," *ACM Trans. Math. Softw.*, vol. 41, no. 2, pp. 11:1–11:36, Feb. 2015. [Online]. Available: http://doi.acm.org/10.1145/2629697

[95] 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

[96] 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.

[97] W. Zhang, A. S. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, "Boxlib with tiling: An adaptive mesh refinement software framework," *SIAM J. Scientific Computing*, vol. 38, no. 5, 2016.

[98] 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 *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, ser. SC '01. New York, NY, USA: ACM, 2001. [Online]. Available: http://doi.acm.org/10.1145/582034.582040 pp. 6–6.

[99] P. H. Worley, A. A. Mirin, A. P. Craig, M. A. Taylor, J. M. Dennis, and M. Vertenstein, "Performance of the community earth system model," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063457 pp. 54:1–54:11.

[100] 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.

[101] 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.

[102] P. MacNeice, K. M. Olson, C. Mobarry, 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

[103] 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.

[104] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[105] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[106] C. Ericson, *Real-Time Collision Detection*. Boca Raton, FL, USA: CRC Press, Inc., 2004.

[107] T. W. Crockett, "An introduction to parallel rendering," *Parallel Comput.*, vol. 23, no. 7, pp. 819–843, July 1997. [Online]. Available: http://dx.doi.org/10.1016/S0167-8191(97)00028-8

[108] B. Schling, "The boost c++ libraries," Tech. Rep., 2011.

[109] 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*, ser. SC '05. IEEE Computer Society, 2005. [Online]. Available: http://dx.doi.org/10.1109/SC.2005.61 pp. 4–.

[110] T. Hoefler and J. L. Traff, "Sparse Collective Operations for MPI," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. IEEE Computer Society, 2009. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5160935 pp. 1–8.

[111] 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*, vol. 33, no. 3, pp. 1103–1133, 2011.

[112] 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*, ser. SC '08. IEEE Press, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413389 pp. 18:1–18:12.

[113] 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)*, S. J. Turner, B. S. Lee, and W. Cai, Eds., 2006, pp. 521–530.

[114] R. E. Tarjan, "Amortized computational complexity," *SIAM Journal on Algebraic Discrete Methods*, vol. 6, no. 2, pp. 306–318, 1985.

[115] D. Mosberger, "Memory consistency models," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 1, pp. 18–26, Jan. 1993. [Online]. Available: http://doi.acm.org/10.1145/160551.160553

[116] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996. [Online]. Available: http://dx.doi.org/10.1109/2.546611

[117] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, pp. 15–26, May 1990. [Online]. Available: http://doi.acm.org/10.1145/325096.325102

[118] G. Quercini and H. Samet, "Uncovering the spatial relatedness in wikipedia," in *Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ser. SIGSPATIAL '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2666310.2666398 pp. 153–162.