Machine and Application Aware Partitioning for Adaptive Mesh Refinement Applications

Milinda Fernando
School of Computing,
University of Utah
Salt Lake City, Utah
milinda@cs.utah.edu

Dmitry Duplyakin
Department of Computer Science
University of Colorado
Boulder, Colorado
dmitry.duplyakin@colorado.edu

Hari Sundar
School of Computing,
University of Utah
Salt Lake City, Utah
hari@cs.utah.edu

ABSTRACT

Load balancing and partitioning are critical when it comes to parallel computations. Popular partitioning strategies based on space filling curves focus on equally dividing work. The partitions produced are independent of the architecture or the application. Given the ever-increasing relative cost of data movement and increasing heterogeneity of our architectures, it is no longer sufficient to only consider an equal partitioning of work. Minimizing communication costs are equally if not more important. *Our hypothesis is that an unequal partitioning that minimizes communication costs significantly can scale and perform better than conventional equal-work partitioning schemes.* This tradeoff is dependent on the architecture as well as the application. We validate our hypothesis in the context of a finite-element computation utilizing adaptive mesh-refinement. Our central contribution is a new partitioning scheme that minimizes the overall runtime of subsequent computations by performing architecture and application-aware non-uniform work assignment in order to decrease time to solution, primarily by minimizing data-movement. We evaluate our algorithm by comparing it against standard space-filling curve based partitioning algorithms and observing time-to-solution as well as energy-to-solution for solving Finite Element computations on adaptively refined meshes. We demonstrate excellent scalability of our new partition algorithm up to 262, 144 cores on ORNL's Titan and demonstrate that the proposed partitioning scheme reduces overall energy as well as time-to-solution for application codes by up to 22.0%.

CCS CONCEPTS

•Computing methodologies →Massively parallel algorithms;

KEYWORDS

domain decomposition; communication minimizing algorithms; energy efficient computing; AMR; FEM

1 INTRODUCTION

As we scale up to exascale machines, the cost of data movement and load-imbalance therein are a major bottleneck for achieving scalability [28] and energy and power efficiency [31]. These are dictated largely by how data (or tasks) are partitioned across processes. This motivates a need for better partitioning schemes for our most demanding applications. While we continue to build larger supercomputers, these come with increased parallelism and heterogeneity at the node as well as the cluster level. In all cases, minimizing load-imbalance and communication-costs as well as minimizing data-movement are critical to ensuring scalability and performance on leadership architectures. Additionally, the partitioning algorithms need to be architecture (*e.g.* bandwidth) and application-aware (*e.g.* data access pattern) and not simply divide the problem into equal-sized chunks across p processes. These are also important for reducing the energy footprint of our algorithms and making extreme-scale computing economically viable [31]. To be practical, such partitioning algorithms need to be optimal, i.e., with a $O(N/p + \log p)$ parallel complexity for partitioning N data across p processes, ideally with low constants [12].

Space Filling Curves (SFC) are commonly used by the HPC community for partitioning data [7, 10, 36] and for resource allocations [3, 32]. By mapping high-dimensional spatial coordinates (or regions) onto a 1D curve, the task of partitioning is made trivial. Locality, i.e., ensuring that regions that are close in the higher dimensional space are also close along the 1D curve, is guaranteed by the *ordering* of the curve, leading to different SFC, such as Morton, Hilbert, Peano, etc. The key challenge is to order the coordinates or regions according to the specified ordering, usually performed using an ordering function and sorting algorithm. This approach is easily parallelized using efficient parallel sorting algorithms such as SampleSort [11]. This is the approach used by several state-of-the-art packages [5, 7, 10, 24, 36]. This approach either aims to get the ideal load balance ($N/p \pm 1$) or relies on the underlying sorting algorithm to load balance. In either case there is no expectation on the minimization of the communication costs beyond what is afforded by the locality of the SFC.

Our hypothesis is that an unequal partitioning that minimizes communication-costs significantly and is architecture and application-aware, can scale and perform better than conventional equal-work partitioning schemes. In this work, we test this hypothesis in the context of SFC-based partitioning and present novel sequential and distributed partitioning algorithms that minimize the overall time-to-solution as well as the energy-to-solution by minimizing communication costs in exchange for a (small) increase in work-load imbalance. We evaluate energy-to-solution in addition to time-to-solution to ensure that the increased load-imbalance does not result in increasing the energy-cost of computations. Our approach

HPDC '17, June 26-30, 2017, Washington, DC, USA

DOI: http://dx.doi.org/10.1145/3078597.3078610

is architecture and application-aware and will produce different partitions on different machines and for different applications.

Related Work. Load balancing and partitioning are critical when it comes to parallel computations. Generally partitioning involves equally dividing the work and data among the processors, reducing processor idle time and communication costs. The standard approach is to model the problem using a communication or data-dependency graph and partition the vertices of the graph into (roughly) equal-sized groups such that the weight of the edges crossing across processes is minimized. Graph partitioning is a NP-hard problem and most work focuses on heuristics to obtain good approximations. Several graph partitioning packages exist [9, 17, 22, 23, 37] but performance and parallel scalability is challenging, especially for applications requiring repeated partitioning, such as Adaptive Mesh Refinement (AMR). In many such cases, SFC are used as a scalable and effective partitioning technique [2, 5]. Note that several recent Gordon-Bell award winners have used SFC-based partitioning techniques for this reason [15, 18, 27, 30]. One of the main advantage of SFC based partitioning is the preservation of geometric locality of objects between processors. Depending on the SFC (i.e. Morton, Moore, Hilbert) that is used for partitioning, the amount of locality preserved differs [2]. Most SFC based partitioning—especially for adaptive meshing—use the Morton ordering that offers a good balance between the quality of partition and the efficiency of implementation.

There is a large literature of Space Filling Curve (SFC) based partitioning schemes. Several works have compared the clustering properties of space filling curves [1, 25] and concluded the superiority of Hilbert curves over the Morton curve. Gunther *et al*[13] demonstrated that using SFCs to build hierarchical data structures such as octrees minimizes data access time. Algorithms have been proposed for computing the (inverse) mapping between one and d-dimensional spaces [2, 6, 7] including indexing schemes for unequal dimension cardinalities [16], resulting in reduced communication costs. Other applications include multi-dimensional data reordering [20] and to speed up sparse matrix-vector multiplications [38] by improving cache-utilization. Several implementations of SFC-based partitioning algorithms are also available including Dendro [36], p4est [5] and Zoltan [8]. A thorough review of SFCs and their applications can be found in [2].

Contributions. While SFCs have been used for partitioning data for a long time and several efficient implementations exist, our algorithms produce better partitions and demonstrate the scalability and efficiency experimentally. *Our main contribution is the development of a new SFC-based partitioning algorithm that enables incorporation of a machine-model to produce high-quality partitions.* Our contributions in detail:

- **Method:** We present an algorithm that allows us to factor in the overall communication costs during SFC-based partitioning. We demonstrate that reducing load-imbalance is accompanied with increasing communication costs when performed in a top-down manner. By using a performance model to determine the tradeoff between work-imbalance and communication costs, we can determine the optimal partition that results in reduced time-to-solution for application codes. Our approach is also comparable in performance and scalability to existing SFC-based partitioning approaches.
- **Experimental Evaluation:** We conduct experiments to demonstrate the efficiency and scalability of our algorithm on ORNL's Titan up to 262,144 cores. We demonstrate that our algorithm reduces runtime by up to 22% while performing Finite Element computations. We also include energy measurements for resulting MATVEC operations on Cloudlab [29] and demonstrate similar 22% savings in energy-to-solution.

Organization of the paper. The rest of the paper is organized as follows. In §2, we give a quick overview of space filling curves and octree-based adaptive meshing. In §3 we describe the new partitioning algorithm, as well as justifications for trading minor load-imbalance for reduced communication costs. In §4, we discuss the experimental setup including the framework for measuring the energy costs related with a finite element simulation. In §5, we present results demonstrating the superiority of the new partitioning algorithm. Finally, we conclude with directions for future work.

2 BACKGROUND

The most common strategy for parallelizing algorithms is to partition data. The primary goal of the partitioning approach is to divide work equally (load-balance) and minimize the communication between processes. Geometry-based heuristics, such as the use of Space filling curves (SFC) are often the only choice when the data is complex and dynamic, such as for adaptive mesh refinement [2]. For this reason, we will primarily use adaptively refined octree meshes for illustrative purposes, although the proposed methods are equally applicable to other spatial partitioning problems, such as resource allocation[3, 32]. SFCs define a one-to-one mapping between an d-dimensional space and one dimensional space. Since, partitioning 1D data is trivial, such an ordering enables simple load-balanced partitioning of data. The minimization of communication is achieved by the clustering properties of the SFC. For applications where the data dependencies are geometrically local, SFC-based partitioning schemes are efficient, scalable and highly effective. We introduce notation used for the rest of the paper in Table 1.

While SFCs are defined primarily for coordinates, they can be easily extended to support regions, such as octants or elements, by having a notion of an *anchor* coordinate, say the smallest corner along all dimensions, and a measure of the size of the region, such as the level of refinement. In the context of octrees, firstly a discretization of the coordinates is specified by the maximum level of refinement, D_{max}. This allows for the coordinates to be stored as unsigned integers of length D_{max} bits. We will consider ordering for 3D regions specified using 4 values, the anchor (x, y, z) and the level $l \in [0, D_{max})$. As previously mentioned, the usual approach has been to define an ordering function using such coordinates and use existing parallel sorting algorithms such as SAMPLESORT. Since it is not straightforward to incorporate the machine-model into a parallel sorting algorithm like SAMPLESORT, we will first present a modified SFC ordering (and therefore partitioning) algorithm that

$comm$	MPI communicator
p	number of MPI tasks in $comm$
r	the task id (MPI Rank)
A	global: input elements or regions
A_r	local : input elements or regions local to task r
A_i	local : elements or regions in the i^{th} bucket
N	global number of elements in A
$n = N/p$	local number of elements in A
l	refinement level of the curve or tree
R_h	SFC based permutation function
W_{max}	max. of work assigned to an individual processor
C_{max}	max. of data communicated
t_w	interconnect slowness (1/bandwidth)
t_s	interconnect latency
t_c	intranode memory slowness (1/ RAM bandwidth)

Table 1: *Notation used in this paper.*

Algorithm 1 TreeSort (Sequential)

Input: A list of elements or regions A, the starting level l_1 and the ending level l_2
Output: A is reordered according to the SFC.

```
1: counts[] ← 0                                   ▷ |counts| = 2^dim, 8 for 3D
2: for a ∈ A do
3:     increment counts[child_num(a)]
4: counts ← R_h(counts)                           ▷ Permute counts using SFC ordering
5: offsets ← scan(counts)
6: A'[] ← empty
7: for a ∈ A do
8:     i ← child_num(a)
9:     append a to A' at offsets[i]
10:    increment offset[i]
11: swap(A, A')
12: if l_1 > l_2 then
13:    for i := 1 : 2^dim do
14:        TreeSort(A_i, l_1 − 1, l_2)            ▷ local sort
```

we will eventually extend to an architecture and application-aware, communication-minimizing partitioning algorithm.

2.1 Modified SFC Ordering

Let us consider the sequential case first. Instead of relying on a comparison based sorting algorithms such as quicksort or SampleSort, one can instead opt to use a non-comparative sorting algorithm, such as the Radix sort [39]. If one considers the Most-Significant-Digit (MSD) Radix sort, then this is equivalent to a top-down construction of quadtrees or octrees, in 2D and 3D respectively. Bucketing the data (re-ordering) based ok the k-th bit of the x, y, and z coordinates, is equivalent to splitting the octree at the k^{th} level. We call this algorithm TreeSort (Algorithm 1). A key difference between the standard Radix sort and TreeSort is that we need to re-order the buckets based on the SFC-curve. The complexity of the algorithm is not affected by the choice of SFCs. In case of the Morton Curve, the ordering is fixed, independent of the level. For cases where the ordering is based on the level, as in the case of Hilbert, these can be applied at this level with an $O(1)$ cost. Another advantage of this approach is that it traverses the tree in a depth-first fashion leading to good locality and cache utilization.

While it might seem like we have simply replaced SampleSort with a Radix sort, there is an important reason for this. Because of the similarity to the octree construction, note that we get progressively closer to the optimal N/p load on each partition. It is this iterative nature of the Radix sort (induced partitions) that makes it appealing over comparison-based approaches like SampleSort.

3 ARCHITECTURE-OPTIMAL PARTITIONING

Two desirable qualities of any partitioning strategy are load balancing, and minimization of overlap between the processor domains. SFC-based partitioning does a very good job in load balancing but does not permit an explicit control on the level of overlap. Ideally, we would like to have a perfectly load-balanced partition that also minimizes the overall communication. But this is usually not possible, especially for non-uniform work distributions such as for adaptively refined meshes. Additionally, it might not be possible to minimize the load-imbalance and overall communication simultaneously. Since the cost of communication across inter-process boundaries depends both on the machine characteristics, say network bandwidth, as well as the application, i.e., the amount of data being exchanged per unit boundary, it is important to consider these aspects while partitioning data. Specifically, since the goal of most parallel codes is to minimize time-to-solution (and possibly energy-to-solution), it is important that the partition balances "equal assignment of work" with "overall communication cost" to achieve these goals. Clearly such a balance is dependent on both the machine-characteristics as well as the application's data-dependencies. In this section, we will incorporate these features into SFC-based partitioning.

Simple partitions, such as those producing relatively cuboid partitions have a smaller overlap compared with more irregular partitions, as might be produced by a SFC-based partitioning algorithm. In [35], we proposed a heuristic partitioning scheme based on the intuition that a coarse grid partition is likely to have a smaller overlap between the processor domains as compared to a partition computed on the underlying fine grid. This algorithm first constructed and partitioned a complete linear octree based on the data (equivalent to a standard SFC-based partitioning). This was followed by coarsening of the ocree and a second weighted partitioning of the coarse octree to get *simpler* partitioning. There are a few shortcomings of this approach. Firstly, this is a heuristic and there are no guarantees that the partition produced will be better than using the standard SFC-based partition. This is mainly due to the reliance on an external sorting function and the bottom-up (coarsening) fashion in which it operates. Secondly, the algorithm considers neither the machine-characteristics nor the application characteristics and will produce the same partition on different machines and for different applications[1]. OptiPart addresses these shortcomings. We will first present the distributed memory version of TreeSort, then demonstrate that decreasing the load-imbalance via this algorithm results in a monotic increase in overall communication costs, allowing users to specify a tolerance and reduce overall communication costs. We then develop a performance model to estimate the optimal tolerance to obtain the best time-to-solution for the specific machine and application.

[1] e.g. for the Poisson equation vs the wave Equation on the same mesh.

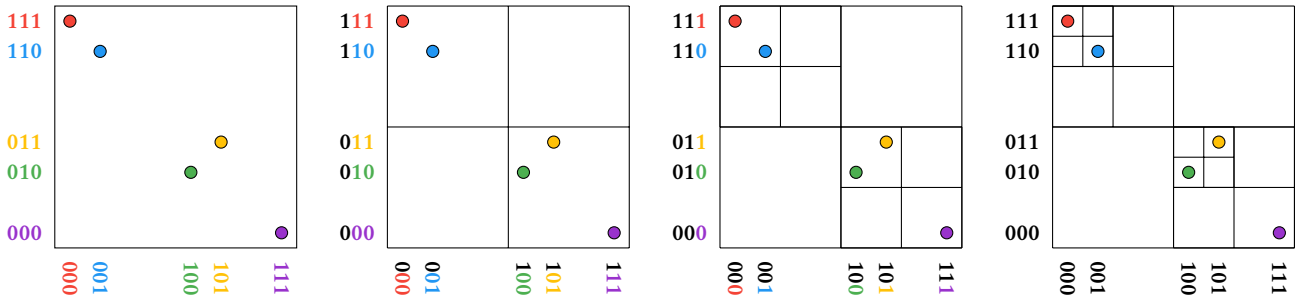

Figure 1: Equivalence of the MSD Radix sort with top-down quadtree construction when ordered according to space filling curves. Each color-coded point is represented by its x and y coordinates. From the MSD-Radix perspective, we start with the most-significant bit for both the x and y coordinates and progressively bucket (order) the points based on these. The bits are colored based on the points and turn black as they get used to (partially) order the points.

3.1 Distributed Memory TreeSort

Our modified SFC-ordering algorithm TreeSort operates in a top-down fashion. In this section, we propose a distributed variant of TreeSort that enables fine control over the load-balance and also enables reduction in communication costs in exchange for higher load-imbalance. The distributed algorithm proceeds as the sequential variant (§2.1), bucketing and partially sorting the data. Unlike the sequential TreeSort, we have to traverse the tree in a breadth first fashion, as the data needs to be distributed across processors. Note that at each level, we split each octant 8 times (for 3D), so in $\log_8 p$ steps we will have p buckets. A reduction provides us with the global ranks[2] of these p buckets. Here p is the desired number of partitions. Using the optimal ranks ($r \cdot N/p$) at which the data needs to be partitioned for process r, we selectively partition the buckets to obtain the correct partitioning of the local data. This is in principle similar to the approach used by algorithms such as histogramsort [33] and hyksort [34], except that no comparisons are needed for computing the ranks. The computational cost corresponds to the $O(N/p)$ bucketing required for each of the $\log_8 p$ levels. We also need p reductions and an all-to-all data exchange. Therefore the expected parallel runtime, T_p for the distributed algorithm is,

$$T_p = t_c \frac{N}{p} + (t_s + t_w p) \log p + t_w \frac{N}{p}, \tag{1}$$

where t_c, t_s, and t_w are the memory slowness (1/RAM bandwidth), the network latency and the network slowness (1/bandwidth), respectively. In our evaluation, we considered trees of depth 30 (so that the coordinates can be represented using `unsigned int`). Note that up to $8^6 = 262,144$ buckets can be determined using six levels, so the cost of determining splitters to distribute the data across processors is significantly lower than other approaches. An analysis of complexities for popular distributed sorting algorithms can be found in [34].

While TreeSort is performed in place, the distributed version requires $O(p)$ additional storage to perform the reduction to compute the global splitter ranks. The additional storage as well as the cost of performing the reductions can be significant for large p, therefore we perform the splitter selection in a staged manner. We limit the maximum number of splitters to a user-specified parameter $k \leq p$. This also reduces the cost of the reduction to compute the global ranks of the splitters from $O(p \log p)$ to $O(k \log p)$. The expected running time for the staged distributed TreeSort is,

$$T_p = t_c \frac{N}{p} + (t_s + t_w k) \log p + t_w \frac{N}{p}. \tag{2}$$

Additionally, the all-to-all exchange is also performed in a staged manner similar to [4, 34], avoiding potential network congestion. We will now develop the algorithm further to automatically determine the best tolerance.

Remark. *We will develop the distributed TreeSort algorithm further to balance work and communication costs based on the machine-model in §3.4. Therefore, for clarity of presentation, we are not presenting the pseudocode for distributed TreeSort. The pseudocode for OptiPart is presented in Algorithm 3.*

3.2 Justification for Flexible Partitioning

An advantage of the distributed TreeSort algorithm is that we can specify a tolerance, *tol*, for the desired load-balance, i.e., stopping if the induced partitions are $r \cdot N/p \pm tolerance$ instead of the optimal $r \cdot N/p$. While it is possible to specify such a tolerance for samplesort variants [21, 33, 34], the advantage is limited to reducing the cost of computing the partition or ordering at the cost of reduced load-balance and will not provide any reduction in communication costs. SFC-based partitioning algorithms are likely to partition using the *finest* level octant, resulting is increased boundary surface, as the primary criterion is to equally divide the work (octants) amongst the processes. Our hypothesis is that, it should be possible to find a partition in close proximity to the optimally load-balanced partition, that has a lower inter-process boundary surface. This will enable users to get the partition with the minimum boundary surface, by specifying a tolerance on the equally-divided load. In case of TreeSort, specifying a tolerance, in addition to making the ordering faster to compute, the induced partition also has reduced communication costs (for subsequent computations, like numerical simulations) in exchange for the increased load-imbalance. This makes TreeSort attractive when the communication costs are high. If we consider the cost of communication to be `10x` that of performing the work on one unit of data, then an increase of 20

[2]The rank here referes to the position of a element in a sorted array.

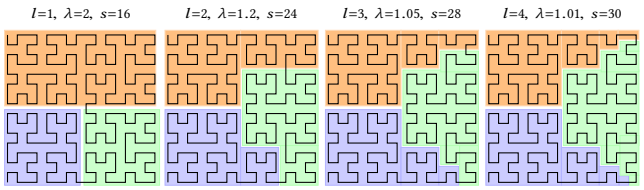

Figure 2: Illustration of the increase in communication costs with increasing levels of TreeSort. Partitions for the case of $p = 3$ are drawn with the boundary of the partition (s) and the load-imbalance (λ) given along with the level (l) at which the partition is defined. At each level, the orange partition (■) gets the extra load that is progressively reduced. The green partition (■) gets the largest boundary that progressively increases.

units of work resulting in a reduction of 5 units of data-exchange, would still provide savings of $5 \times 10 - 20 = 30$ units. This is a contrived example, but the key point is that even small reductions in data-movement over the network provide large savings in overall runtime.

By design, for the TreeSort algorithm the load-imbalance, $\lambda = \max(|W_r|)/\min(|W_r|)$ decreases with increasing l getting closer to the optimal value of $\lambda = 1$. However, as we increase levels, the boundary of the partition s is non-decreasing. This is illustrated in Figure 2 using a simple 2D partition using 3 processors. This allows the user to specify a tolerance, say 1%, which when reached will prevent further refinement, potentially reducing the inter-process boundary and thereby the communication costs of subsequent operations. Note that the claim is not for reducing the data-exchange cost during the reordering, but for subsequent operations that might be performed based on the partition.

The example in Figure 2 considers uniform refinement, but the result is also true for meshes with adaptive refinement, as would be the case for most numerical simulations. This is demonstrated in Figure 3. Here we consider the partition between two processors, and the specific element that will be refined at the next level. It can be seen that for all cases except one, the surface area of the partition is non-decreasing. The case where the surface area decreases is a case of extreme refinement, that will only occur if the last child has a significantly higher refinement compared to the other siblings. While this case appears to limit the effectiveness of the approach, it is important to realize that other more-expensive approaches like spectral bisection also fail for similar examples [26].

3.3 Performance Model

While having the user specify a tolerance for the load-balance in order to lower the communication costs allows for better performance, it does limit the portability of the method and makes it difficult when either the architecture or the data distribution changes. Ideally, we would like to automatically determine the optimum tolerance based on the application and the machine characteristics. As mentioned previously, the tradeoff is between the load and the communication costs across all machines. Additionally, the time for either stage will be dominated by the processor that has

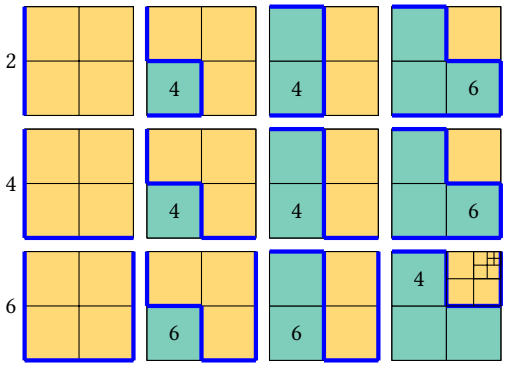

Figure 3: Demonstration that the communication costs are non-decreasing with increasing levels of TreeSort for most refinements and identification of the pathological case where the surface area decreases (bottom-right). The left column corresponds to initial boundaries (blue line) sharing 1, 2, and 3 faces of the quadrant (rows) that will be refined at level l. The remaining columns illustrate the change in surface area for the cases where 1-3 child nodes get added to the blue partition. The numbers represent the surface area of each case.

the maximum load (W_{max}) or has to communicate the maximum (C_{max}) amount of data. We build a simple performance model to characterize the overall parallel runtime as a combination of these two terms. We further note that these times can be estimated using the network bandwidth ($1/t_w$) and the memory bandwidth ($1/t_c$)– for memory-bound computations. In other words the total runtime (T_p) can be characterized by the following equation,

$$T_p = \alpha t_c W_{max} + t_w C_{max}. \tag{3}$$

Here, α is indicative of the number of memory accesses performed per unit of work. For example, if the target application is a 7-point stencil operation, then α will be ~ 8. In general, this can be computed using a simple sequential profiling of the main execution kernel. While this model ignores aspects such as overlapping computation and communication, it is simple and can help us easily determine if given a set of partitions with different W_{max}, C_{max} which partition is likely to be the most efficient.

Remark. *It is easy to modify (3) for a compute-bound application, and even use a simple profiling step to determine the right parameters for running a specific application on a specific architecture.*

We would also like to briefly discuss the model from the energy perspective, as there might be concerns that the increased load imbalance might increase the overall energy cost of the computation. For most modern cluster architectures, the overall energy will be strongly correlated with the overall runtime. Assuming an efficient processor architecture, the overall energy for the computation will not depend on the partitioning, as the sum of *work* will remain the same. The overall energy cost of communication will however be lower for the lowest total data communicated. This is what we aim for, i.e., a partition that gives the best runtime by balancing W_{max} and C_{max} and minimizes the total energy required

Algorithm 2 PartitionQuality

Input: A distributed list of elements or regions A_r, *comm*, splitters s,
Output: Predicted execution time T_p for current splitters s

$bdyOctants \leftarrow computeLocalBdyOctants(A_r, s)$
$localSz \leftarrow size(A_r, s)$
MPI_ReduceAll($bdyOctants$, C_{max}, MPI_MAX, $comm$)
MPI_ReduceAll($localSz$, W_{max}, MPI_MAX, $comm$)
$T_p \leftarrow \alpha t_c W_{max} + t_w C_{max}$
return T_p

for the computation by additionally minimizing C_{max}. However, since energy-to-solution is an increasingly important metric for current and future HPC systems, we will also analyze the energy-to-solution in addition to time-to-solution while evaluating our new partitioning algorithm.

3.4 OptiPart: Architecture & Data optimized partitioning

Armed with our performance model, we can easily modify the distributed TreeSort to compute the optimal partition without having to guess the appropriate tolerance. We will use the memory and network slowness (t_c, t_w) based on the machine and will expect the user to provide the parameter α that is representative of the core computations. We call this algorithm OptiPart. The algorithm proceeds the same as distributed TreeSort , but instead relies on estimates of W_{max}, C_{max} for the current and next refinements and proceeds only if the runtime estimates (3) for the next refinement are lower than the current estimate. This does not change the complexity of the partitioning algorithm, requiring only $O(N/p)$ local work and a single reduction $O(\log p)$. OptiPart relies on a helper routine that computes the quality of the current partition, by doing a linear pass over the elements to determine the size of the local boundary. The pseudocode for this routine is given in Algorithm 2. Finally, the pseudocode for OptiPart is given in Algorithm 3. Note that the standard distributed TreeSort can be recovered by iterating till the work is equally divided instead of using Algorithm 2 to estimate the partition quality.

4 EXPERIMENTAL SETUP

Large scalability experiments reported in this paper were performed on Titan and Stampede. Titan, a Cray XK7 supercomputer at Oak Ridge National Laboratory (ORNL), has a total of 18,688 nodes consisting of a single 16-core AMD Opteron 6200 series processor, for a total of 299,008 cores. Each node has with 32GB of memory. It is also equipped with a Gemini interconnect and 600 terabytes of memory across all nodes. Stampede at the Texas Advanced Computing Center (TACC), is a linux cluster consisting of 6400 computes nodes, each with dual, eight-core processors for a total of 102,400 available cpu-cores. Each node has two eight-core 2.7GHz Intel Xeon E5 processors with 2GB/core of memory and a 3 level cache. Stampede has a 56Gb/s FDR Mellanox InfiniBand network connected in a fat tree configuration. In our largest runs we use a total of 262,144 cores on Titan.

4.1 Power Measurements

In order to quantify energy consumption tradeoffs, we provisioned two clusters on the CloudLab testbed [29]: *Wisconsin-8* – an 8-node

Algorithm 3 OptiPart

Input: A distributed list of elements or regions A_r, *comm*, p (w.l.g., assume $p = mk$), r of current task in *comm*,α,t_c,t_w,
Output: globally sorted array A

$counts\_local[] \leftarrow 0$, $counts\_global[] \leftarrow 0$
$s \leftarrow$ TreeSort(A_r, $l - \log(p)$, l) ▷ initial splitter computation
$default \leftarrow PartitionQuality(A_r, comm, s)$
$current \leftarrow default$
while $default \geq current$ **do**
  $counts[] \leftarrow 0$ ▷ $|counts| = 2^{dim}$, 8 for 3D
  for $a \in A_r$ **do**
    increment $counts[child\_num(a)]$
  $counts\_local \leftarrow push(counts)$
  $counts \leftarrow R_h(counts)$ ▷ Permute counts using SFC ordering
  offsets $\leftarrow scan(counts)$
  $A'[] \leftarrow empty$
  for $a \in A_r$ **do**
    $i \leftarrow child\_num(a)$
    append a to A' at offsets$[i]$
    increment offset$[i]$
  $swap(A_r, A')$
  MPI_ReduceAll($counts\_local$, $counts\_global$, MPI_SUM, $comm$)
  $s \leftarrow select(s, counts\_global)$
  $default \leftarrow current$
  $current \leftarrow PartitionQuality(A_r, comm, s)$
MPI_AlltoAllv(W_r, s, $comm$) ▷ Staged All2all
TreeSort(A_r, 0, l) ▷ local sort

cluster in the Wisconsin datacenter consisting of nodes with 2x Intel E5-2630 v3 8-core Haswell CPUs (2.40 GHz), 128GB ECC Memory, and 10Gb Ethernet NICs, and *Clemson-32* – a 32-node cluster in the Clemson datacenter with each node having 2x Intel E5-2683 v3 14-core Haswell CPUs (2.00 GHz), 256GB ECC Memory, and a 10Gb Ethernet NIC. We configured the provisioned hardware into SLURM-based [32] cluster environments and ran a set of selected parallel jobs with 256 (on Wisconsin-8) and 1792 (on Clemson-32) MPI tasks.

During execution, we obtained on-board IPMI sensor information and recorded every machine's instantaneous power draw (in Watts) every second. As concluded in [14] and stated in a recent survey [19], power samples collected using IPMI are accurate enough as long as the temporal load-varying effects do not occur at the rate that is near the sampling rate. Accurate energy estimation is a difficult task for short-period jobs. The energy experiments reported in this work includes over 380 jobs that are between 2 and 14 minutes in duration–120 to 8400 samples–on these clusters. In our evaluation, with the aforementioned runtimes and the number of power samples we collected, we are convinced that we are able to draw reliable quantitative conclusions about the energy tradeoffs.

After job completion, we combined the recorded power traces with the job start and end timestamps from the scheduler and obtained per-job energy consumption estimates (in Joules). In addition to the total job consumption, we estimated the amount of energy consumed during the communication phase (i.e. matvec operation in FEM computations) of these jobs. In order to eliminate the impact of the dynamic CPU frequency scaling on our energy estimates, we disabled the dynamic scaling and set all CPU cores to run at 2.4 and 2.0 GHz on Wisconsin-8 and Clemson-32, respectively.

4.2 Implementation details

All algorithms described in this work were implemented using C++ using MPI. We tested the performance using randomly generated

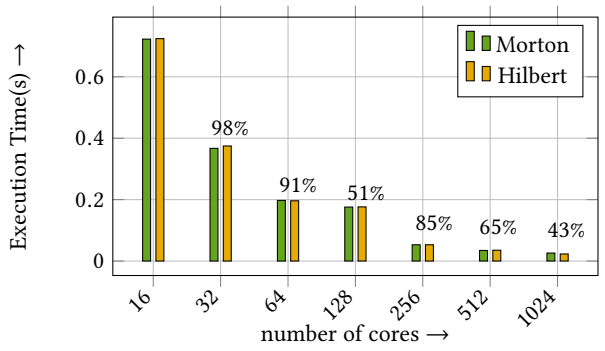

Figure 4: Strong scaling results for Hilbert & Morton based partitioning using a problem size of 16×10^6 elements on ORNL's Titan using core-counts from 16 to 1024. The parallel efficiency for each case (rounded) is listed above the bars.

octrees according to three distributions, uniform, normal, and log-normal. These were generated using the standard c++11 random number generators. No significant difference in performance was observed across the distributions, and therefore the distribution information has been suppressed for clarity of presentation. All results presented in this paper are for data generated according to the normal distribution. We implemented standard FEM computation on randomly generated octrees, focusing on matrix vector multiplication (matvec) for energy measurements. All the energy measurements are done in CloudLab (Wisconsin-8 & Clemson-32) cluster and all the other performance runs are done on ORNL's Titan and TACC's Stampede supercomputers.

5 PERFORMANCE EVALUATION

In this section we present results demonstrating the scalability and performance of OptiPart as well as the quality of the partitions produced. We will first demonstrate the scalability of OptiPart and its capability of partitioning large adaptive meshes efficiently. We will also compare its performance with standard SFC-based partitioning algorithms, primarily to demonstrate that OptiPart's performance is comparable to existing approaches. This is followed by detailed characterization of the partitioning qualities of the partitions generated by OptiPart in the context of a finite element computations on adaptively refined meshes.

5.1 Scalability

In this section we present both strong and weak scalability results carried out for OptiPart. Figure 4 shows the strong scalability results, using a fixed problem size of 16×10^6 elements, with increasing number of cores on ORNL's Titan. We present results for two popular space filling curves, Hilbert and Morton [2], to demonstrate that our algorithm is insensitive to the choice of the SFC. We obtain very good strong scalability with a parallel efficiency of $\sim$ 43% for a 64x scaleup. We are able to partition 16 million elements in $\sim$ 25 msecs across 1024 cores on Titan. We also present weak scalability results, in figure 5 up to 262, 144 cores on ORNL's Titan. We used a grain size of 10^6 elements per process for 16 to 262, 144 processes. In our largest run, we were able to partition 262 Billion elements in $\sim$ 4 seconds across 262, 144 processes. Note that the increase in runtime is largely due to the increased cost of moving elements across the network (`MPI_Alltoallv`) whereas the partitioning algorithm demonstrate better scalability.

5.2 Comparison with SFC-based partitioning

In this section, we present a comparison between existing SFC-based partition schemes and OptiPart. Most existing SFC-based partitioning algorithms rely on parallel sorting algorithms such as SampleSort along with an ordering defined based on the SFC. In general, we would expect OptiPart to be slightly faster than a comparable implementation of SFC-partitioning relying on parallel sorting, as we terminate the splitter selection early. In this comparison, our primary goal is to demonstrate that incorporating the machine and application model does not adversely affect the efficiency or scalability of OptiPart compared to standard SFC-based partitioning. We compare against the SFC-based partitioning implemented in Dendro[36]. This implementation uses the Morton ordering along with SampleSort to partition data. This implementation was also used by the 2010 Gordon-Bell winner [27], and has been demonstrated to scale to leadership architectures. Since OptiPart produces different partitions for different machines, we performed this comparison on both ORNL's Titan as well as TACC's Stampede supercomputers using a grain-size of 5×10^6 elements. These results are presented in Figure 6. We can see that OptiPart has a small performance and scalability improvement over Dendro. This could be due to implementation differences, but the major take-away from this experiment is that it is possible to get application and architecture-aware partitioning without sacrificing performance or scalability.

5.3 Test application

Our target applications are solving Partial Partial Differential Equations (PDEs) using adaptive discretizations using the Finite Element method (FEM). In most computational codes, the basic building block is the matvec, a function that takes a vector and returns another vector, the result of applying the discretized PDE operator to the the input vector. Complex operations such an non-linear operators, time-dependent problems, and using iterative solvers to solve a linear system can all be represented as a series of matvecs. The communication as well as the compute pattern for most PDEs is characterized by the matvec. For this reason, we evaluate the effectiveness of OptiPart using a adaptively discretized Laplacian operator. This is equivalent to us solving a 3D Poisson problem with zero Dirichlet boundary conditions on a unit cube. In order to avoid any issues related to the convergence, we run all application comparisons using 100 matvecs using the standard partition produced by Dendro as well as the partition produced by OptiPart.

5.4 Improved performance for the matvec

In this section we present experiments demonstrating the reduction in time-to-solution and energy-to-solution for 100 iterations of the Laplacian operator using the OptiPart algorithm. All energy measurements were carried out in Clemson-32 (1792 cores) and Wisconsin-8(256 cores) in CloudLab cluster. Figure 7 shows the

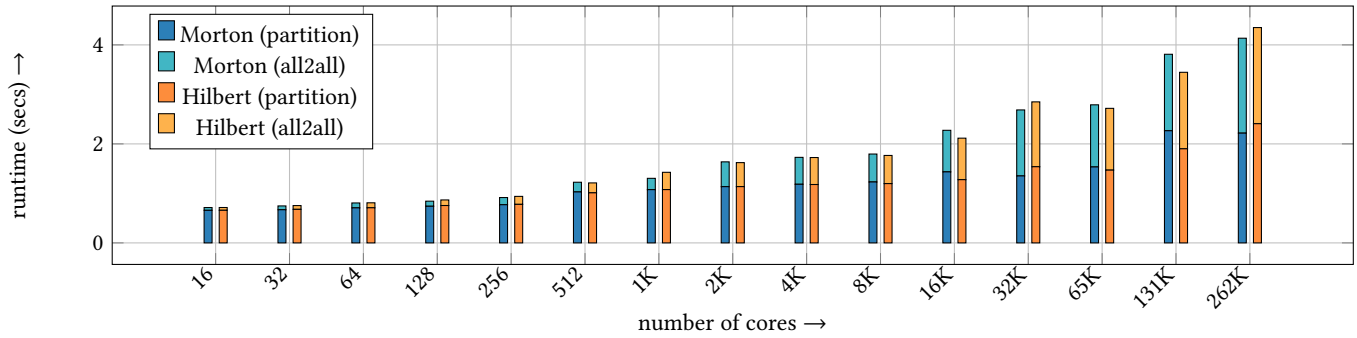

Figure 5: Total execution time for Hilbert & Morton curve based partitioning scheme with a grain size of 10^6elements (minimum of $16M$ & maximum of $262B$ elements), on ORNL's Titan with number of cores from 16 to $262,144$. The total time is divided into time for computing the partition (partition) and the cost of actually exchanging data (all2all).

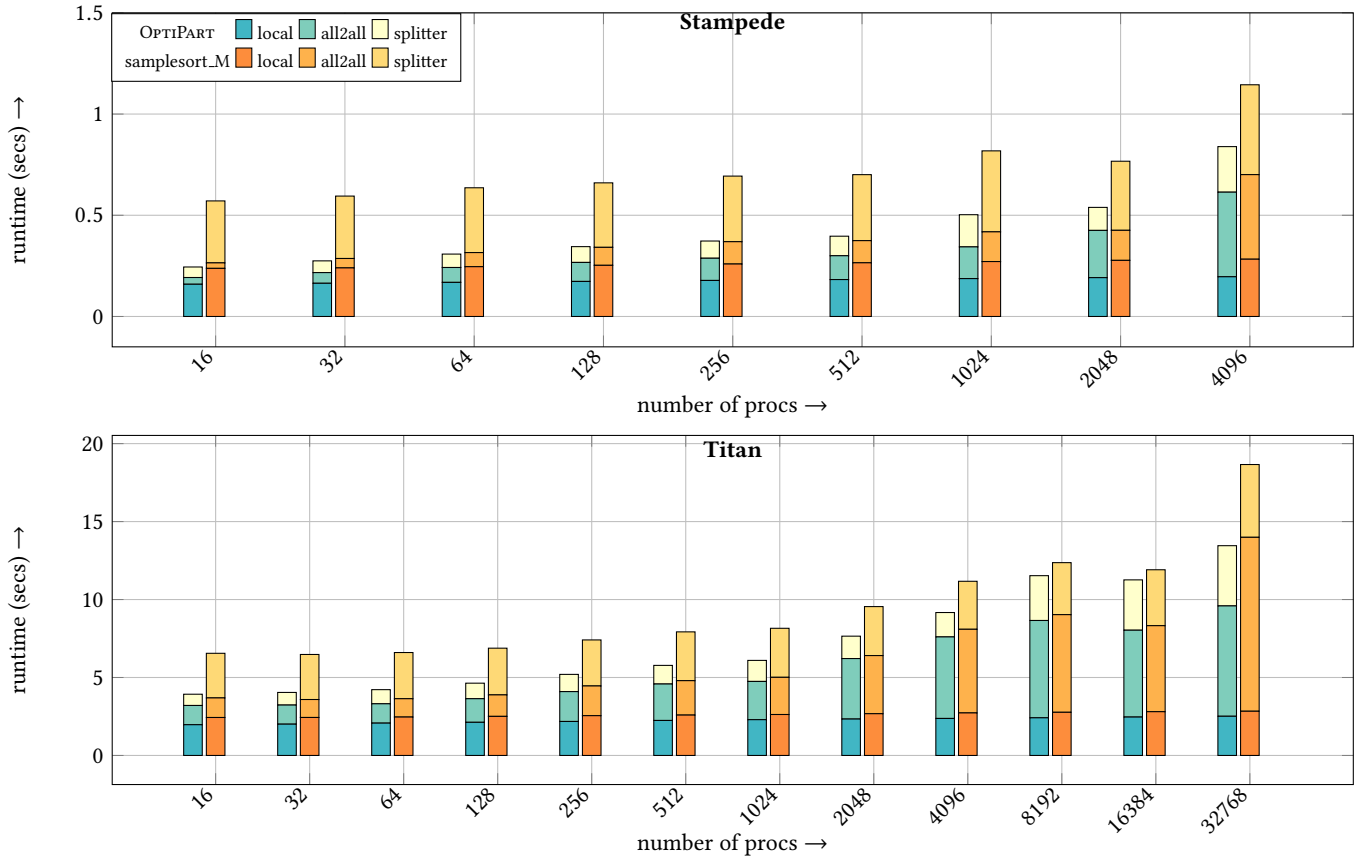

Figure 6: Weak scaling results breakdown (local sort , all to all , splitter calculation) for OptiPart & SampleSort in Dendro[36] with a grain size of 10^6 octants on Stampede (top) and on Titan (bottom) with a grain size of 5×10^6 octants. The scalability of OptiPart is better than SampleSort as show in above figure. Note that the performance and partitions produced by OptiPart are architecture specific, hence the differences between the results on Stampede and Titan.

runtime and energy consumption using an adaptive discretization with $179.2M$ elements on 1792 cores for different tolerances. Note that the energy and runtime are strongly correlated. Additionally, the Hilbert curve produces better partitions compared to the Morton curve. Both curves show reduction in runtime as well as energy for tolerances > 0, indicating that our initial hypothesis is correct. Figure 8 shows the runtime and energy consumption of the same matvec operation for a smaller problem size of $95M$ octants on 256

cores (8 nodes). A closer look at the energy consumption across these 8 nodes for the best tolerance (0.3), shown in Figure 9, that while there is some variability in the energy consumption across the nodes, there is reduction in energy across all nodes. This is true for both the Hilbert and the Morton curves.

Optimal tolerance. We use the performance model, §3.3, to estimate the optimal tolerance. In order to validate our model and determine if it indeed was able to obtain the best partition, we compared our predicted runtime with the actual runtime using a brute-force run for all tolerance values. These results are presented in Figure 10 for the HILBERT curve. This comparison suggests that the overall time consumed by MATVEC operation is directly correlated with the maximum work assigned to an individual processor (W_{max}), the maximum amount of data exchanged for any two processors (C_{max}), the underlying architectural (t_c, t_w) parameters and the computational kernel of the application (α). Please note that OPTIPART starts from a higher tolerance and progressively decreases this, i.e. in the plot it approaches the optimum from the right. For this example, OPTIPART will not reduce the tolerance below 0.3 as the predicted runtime increases at this stage and will terminate the partition.

5.5 Quality of induced partitions

Our main motivation for developing architecture-aware partitioning algorithms, is to lower the communication costs and consequently reduce the time-to and energy-to solution for application codes. In this section, we analyze the partitions produced by OPTIPART to understand the savings better. For FEM computations, processes need read-only access to information from neighboring processes–commonly referred as ghost/halo regions–in order to perform the MATVEC. The performance and scalability of the parallel code depends on both the number of processes that a process needs to communicate with as well as the total amount of data that it needs to send or receive. The communication pattern can be represented in the form of a communication matrix $\mathcal{M}$, where

$$\mathcal{M} = \begin{cases} m_{ij} & \text{if } p_i \text{ needs access to } m_{ij} \text{ elements on } p_j \\ 0 & \text{if no shared data between } p_i \text{ and } p_j \end{cases}$$

We can consider the number of non-zeros (NNZ) elements in the communication matrix $\mathcal{M}$ as a metric for communications cost between partitions since it captures the total number of messages that are exchanged during the computation. We use total amount of data communicated between partitions as another metric. We collect these metrics for a fixed problem size (fixed number of elements and number of processors) with varying tolerance value and evaluate how above mentioned metrics behave for the same input data.

Number of non-zeros (NNZ): We observed the effect of varying tolerance on the number of non-zeros in the communication matrix for $1B$ elements partitioned across 4096 processes. These results are shown in Figure 12. Note the scale difference of two graphs in figure 12 which is due to higher locality preserving nature of the HILBERT curve compared to the MORTON curve. Figure 12 suggests that NNZ strictly decreases with increasing tolerance value, but note that although we can reduce the NNZ with increasing tolerance value, this leads to increased load & communication imbalances (see Figure 11). In order to get optimal partitions (in terms of communication and energy) we need to find the tolerance value that does not disproportionately affect the work imbalance.

Total data communicated: Total data communicated during MATVEC operation in FEM computations is directly coupled with the overall communication cost of the partitions. Figure 12 shows the total data exchanged during 100 MATVEC operations for a fixed problem size on the CloudLab cluster. As expected we can reduce the total data exchanged by increasing the flexibility in the OPTIPART implementation. Figure 12 demonstrates the superiority of HILBERT compared to MORTON in terms of communication cost. The results for HILBERT empirically confirm our observations that the communication decreases with increasing tolerance. The kink in MORTON in Figure 12 is likely due to the discontinuous partitions that are possible with MORTON ; a case we did not consider in our discussions in §3.2.

6 CONCLUSIONS & FUTURE WORK

In this work we presented a new partitioning algorithm that by being architecture and application aware is able to reduce parallel runtime as well as overall energy consumption. The key idea is to assign unequal work to processes in order to reduce overall communication costs. By incorporating machine characteristics such as the slowness of memory and network as well as the applications characteristics we were able to develop a performance model that is able to predict the optimal tradeoff between reducing communication costs and increased load-imbalance. We demonstrated the scalability of the proposed partitioning algorithm up to 262, 144 cores on ORNL's Titan Supercomputer. We also demonstrated energy savings of up to 22% while using the new partition compared to standard SFC-based partitioning algorithms for performing FEM based MATVEC. Our code is available on github[3] so that other researchers can incorporate these methods in their codes. For future work, we would like to refine our performance model with additional information about the machine and the application, such as NUMA and memory access patterns. While the current work is developed in the context of SFC-based partitioning algorithms, the key ideas are applicable to other partitioning algorithms and will be the focus of future research. Specifically, we are working on incorporating architecture and application models while partitioning irregular applications.

ACKNOWLEDGMENTS

We thank the reviewers whose feedback greatly improved this paper. This work was supported in part by the National Science Foundation grants ACI-1464244 and CCF-1643056. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725 and the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant ACI-1548562.

[3] https://github.com/orgs/paralab

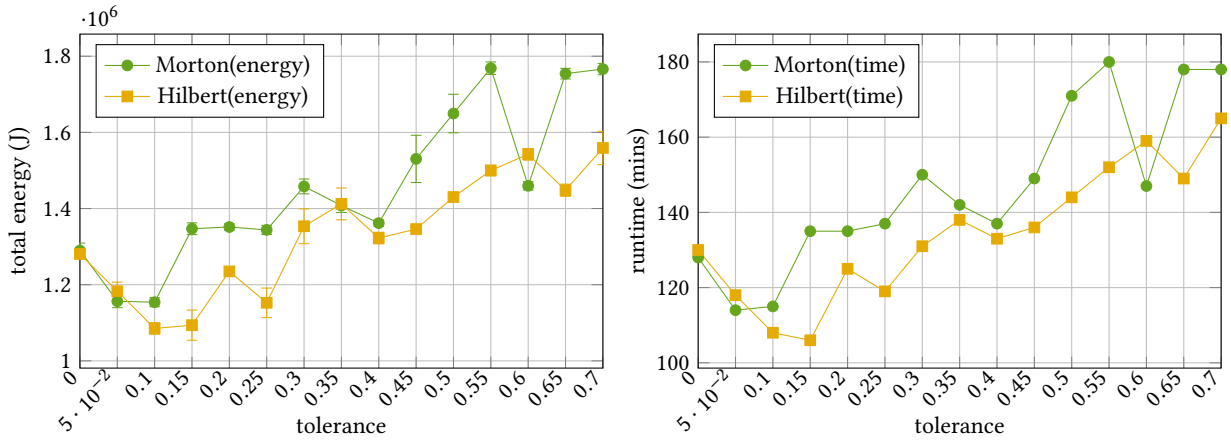

Figure 7: (left) Total energy & time consumption (right) for 100 iterations of MATVEC(distributed) operations, for HILBERT and MORTON curve based partitioning with initial element grain size with 10^5 with maximum depth (of octree) of 30 across 1792 MPI tasks on the Clemson CloudLab cluster.

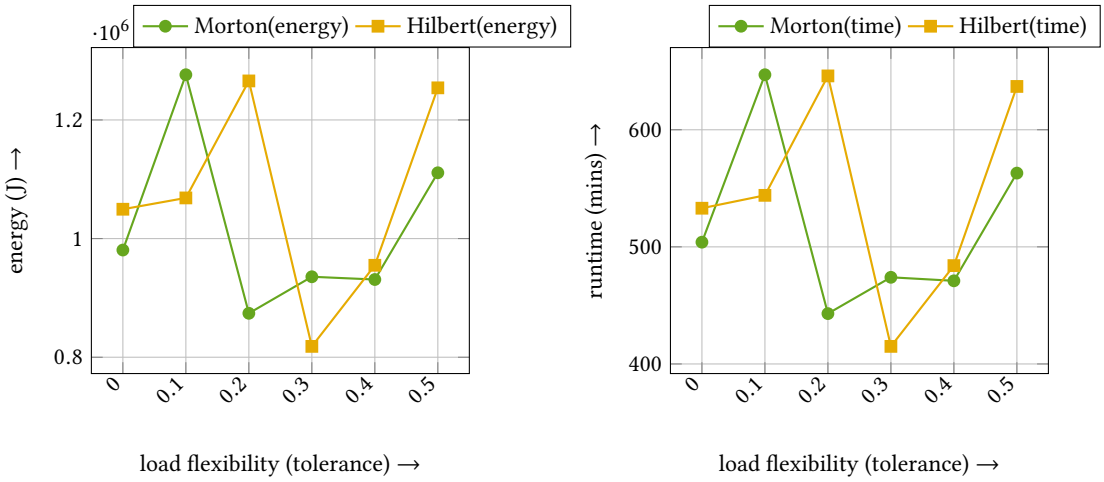

Figure 8: Comparison for MATVEC energy consumption based on HILBERT and MORTON based partitioning schemes for a mesh size of 95M nodes with 256 mpi with varying tolerance values in CloudLab Wisconsin cluster.

REFERENCES

[1] David J Abel and David M Mark. 1990. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information System* 4, 1 (1990), 21–31.

[2] Michael Bader. 2012. *Space-filling curves: an introduction with applications in scientific computing*. Vol. 9. Springer Science & Business Media.

[3] Michael A Bender. 2006. Compute Process Allocator (CPA). *Urbana* 51 (2006), 61801.

[4] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *Parallel and Distributed Systems, IEEE Transactions on* 8, 11 (1997), 1143–1156.

[5] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. 2011. p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing* 33, 3 (2011), 1103–1133. DOI: http://dx.doi.org/10.1137/100791634

[6] Arthur R Butz. 1971. Alternative algorithm for Hilbert's space-filling curve. *IEEE Trans. Comput.* 4 (1971), 424–426.

[7] Paul M Campbell, Karen D Devine, Joseph E Flaherty, Luis G Gervasio, and James D Teresco. 2003. Dynamic octree load balancing using space-filling curves. *Williams College Department of Computer Science Technical Report CS-03* 1 (2003), 68.

[8] U.V. Catalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy, and L.A. Riesen. 2007. Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations. In *Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE. Best Algorithms Paper Award.

[9] Cédric Chevalier and François Pellegrini. 2008. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing* 34, 6 (2008), 318–331.

[10] Karen D Devine, Erik G Boman, Robert T Heaphy, Bruce A Hendrickson, James D Teresco, Jamal Faik, Joseph E Flaherty, and Luis G Gervasio. 2005. New challenges in dynamic load balancing. *Applied Numerical Mathematics* 52, 2 (2005), 133–152.

[11] W. D. Frazer and A. C. McKellar. 1970. Samplesort: A Sampling Approach to Minimal Storage Tree Sorting. *J. ACM* 17, 3 (July 1970), 496–507. DOI: http://dx.doi.org/10.1145/321592.321600

[12] Ananth Grama. 2003. *Introduction to parallel computing*. Pearson Education.

[13] Frank Günther, Miriam Mehl, Markus Pögl, and Christoph Zenger. 2006. A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. *SIAM Journal on Scientific Computing* 28, 5 (2006), 1634–1650.

[14] Daniel Hackenberg, Thomas Ilsche, Robert Schöne, Daniel Molka, Maik Schmidt, and Wolfgang E Nagel. 2013. Power measurement techniques on standard compute nodes: A quantitative comparison. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 194–204.

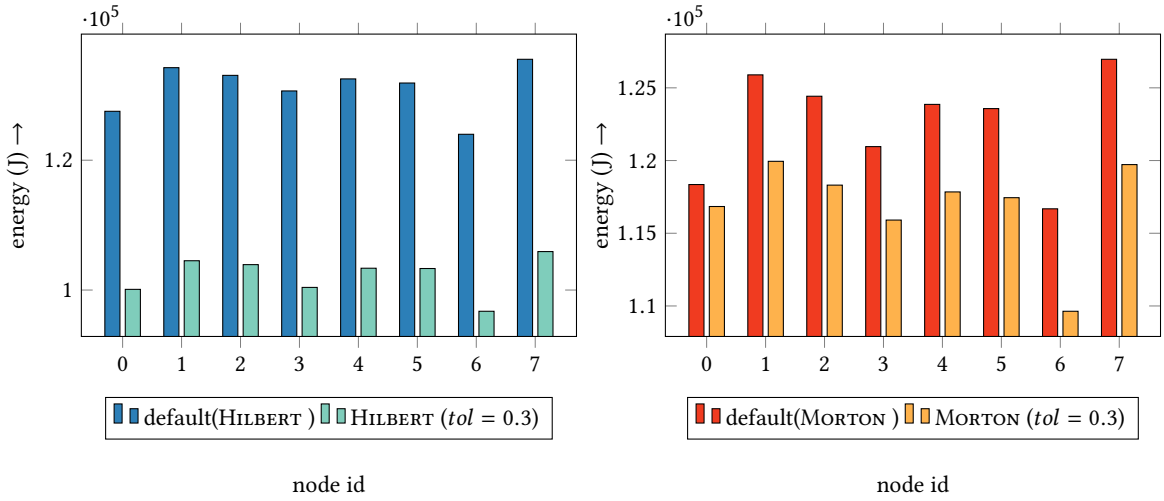

Figure 9: Energy consumed by each node while performing MATVECoperation , with ideal load balancing (for both HILBERT (left) and MORTON (right)) Vs. flexible load balancing with a tolerance of 0.3 for 95M mesh nodes with 256 mpi tasks in CloudLab8 node cluster.

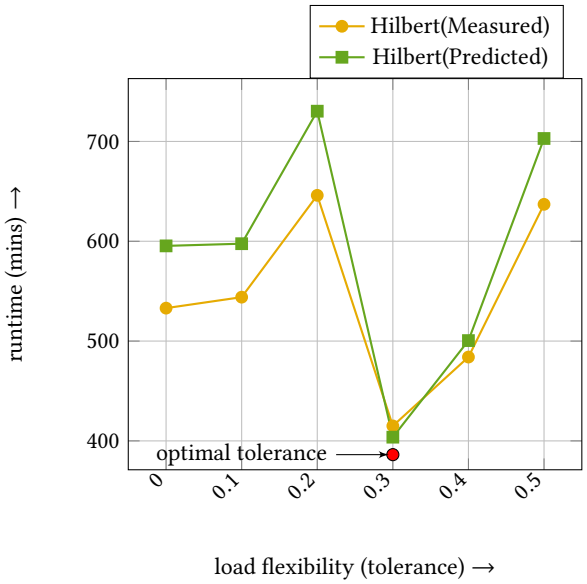

Figure 10: Total time consumed by the 100 MATVEC operations with 256 cores in Wisconsin CloudLab cluster and the interpolated execution time values for HILBERT and MORTON based partitioning, using the model $T_p = \alpha t_c W_{max} + t_w * C_{max}$. This implies the total time consumed during the MATVEC operation directly correlated with maximum amount of work assigned for each core and maximum amount of communication that each core has to carry out. The optimal tolerance that is computed by OPTIPART is highlighted in each figure. Note that OPTIPART starts from a higher tolerance and progressively decreases this, i.e. in the plot it approaches the optimum from the right.

[15] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 2009. 42 TFlops Hierarchical N-body Simulations on GPUs with Applications in Both Astrophysics and Turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*. ACM, New York, NY, USA, Article 62, 12 pages. DOI: http://dx.doi.org/10.1145/1654059.1654123

[16] Chris H. Hamilton and Andrew Rau-Chaplin. 2008. Compact Hilbert Indices: Space-filling Curves for Domains with Unequal Side Lengths. *Inf. Process. Lett.* 105, 5 (Feb. 2008), 155–163. DOI: http://dx.doi.org/10.1016/j.ipl.2007.08.034

[17] Bruce Hendrickson and Robert Leland. 1995. *The Chaco userfis guide: Version 2.0.* Technical Report. Technical Report SAND95-2344, Sandia National Laboratories.

[18] Tomoaki Ishiyama, Keigo Nitadori, and Junichiro Makino. 2012. 4.45 Pflops Astrophysical N-body Simulation on K Computer: The Gravitational Trillion-body Problem. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 5, 10 pages. http://dl.acm.org/citation.cfm?id=2388996.2389003

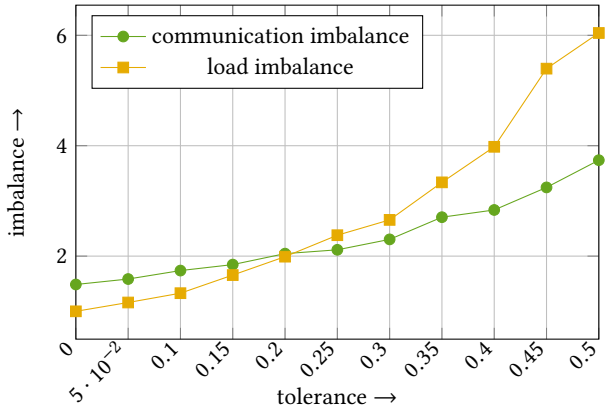

Figure 11: Load imbalance (work_max/work_min) and communication imbalance (bdy_max/bdy_min) plots for HILBERT curve based partitioning, with initial element grain size with 10^5 with maximum depth (of octree) of 30 across 1792 MPI tasks on the Clemson CloudLab cluster.

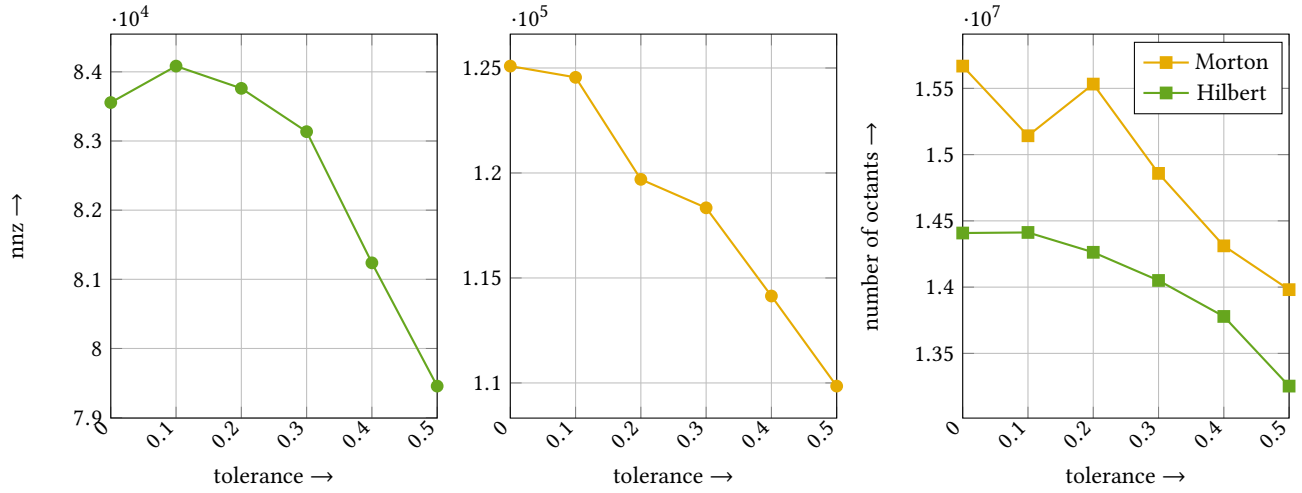

Figure 12: Comparison for number of non-zeros (nnz) elements in the communication matrix corresponding to perform MATVECoperation based on HILBERT (left) and MORTON (center) based partitioning schemes for a mesh size of 1B nodes with 4096 mpi tasks with varying tolerance values. Note that the scale difference between the axes in the plots, and for both partitioning schemes we can reduce the nnz (overall communication cost) by increasing the tolerance value. (right) Total amount of data communicated while performing 100 iterations of MATVEC in Wisconsin-8 in CloudLab with $25.6M$ elements and 256 cores with varying tolerance value.

[19] Chao Jin, Bronis R de Supinski, David Abramson, Heidi Poxon, Luiz DeRose, Minh Ngoc Dinh, Mark Endrei, and Elizabeth R Jessup. 2016. A survey on software methods to improve the energy efficiency of parallel computing. *International Journal of High Performance Computing Applications* (2016), 1094342016665471.

[20] Guohua Jin and John Mellor-Crummey. 2005. Using space-filling curves for computation reordering. In *Proceedings of the Los Alamos Computer Science Institute Sixth Annual Symposium*.

[21] L.V. Kale and S. Krishnan. 1993. A comparison based parallel sorting algorithm. In *International Conference on Parallel Processing, 1993*, Vol. 3. IEEE, 196–200.

[22] George Karypis and Vipin Kumar. 1995. Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0. (1995).

[23] George Karypis, Kirk Schloegel, and Vipin Kumar. 2003. Parmetis. *Parallel graph partitioning and sparse matrix ordering library. Version* 2 (2003).

[24] Justin Luitjens, Martin Berzins, and Tom Henderson. 2007. Parallel space-filling curve generation through sorting. *Concurrency and Computation: Practice and Experience* 19, 10 (2007), 1387–1402.

[25] Bongki Moon, H.v. Jagadish, Christos Faloutsos, and Joel H. Saltz. 2001. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE Transactions on Knowledge and Data Engineering* 13, 1 (2001), 124–141. DOI:http://dx.doi.org/10.1109/69.908985

[26] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. 1990. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM J. Matrix Anal. Appl.* 11, 3 (1990), 430–452. DOI:http://dx.doi.org/10.1137/0611030 arXiv:http://dx.doi.org/10.1137/0611030

[27] Abtin Rahimian, Ilya Lashuk, Shravan Veerapaneni, Aparna Chandramowlishwaran, Dhairya Malhotra, Logan Moon, Rahul Sampath, Aashay Shringarpure, Jeffrey Vetter, Richard Vuduc, Denis Zorin, and George Biros. 2010. Petascale Direct Numerical Simulation of Blood Flow on 200K Cores and Heterogeneous Architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. DOI: http://dx.doi.org/10.1109/SC.2010.42

[28] DOE report. 2010. The opportunities and challenges of exascale computing. (2010).

[29] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE* 39, 6 (2014), 36–38.

[30] Johann Rudi, A. Cristiano I. Malossi, Tobin Isaac, Georg Stadler, Michael Gurnis, Peter W. J. Staar, Yves Ineichen, Costas Bekas, Alessandro Curioni, and Omar Ghattas. 2015. An Extreme-scale Implicit Solver for Complex PDEs: Highly Heterogeneous Flow in Earth's Mantle. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 5, 12 pages. DOI:http://dx.doi.org/10.1145/2807591.2807675

[31] John Shalf, Sudip Dosanjh, and John Morrison. 2011. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*. Springer, 1–25.

[32] Leszek Sliwko and Vladimir Getov. 2015. Workload Schedulers-Genesis, Algorithms and Comparisons. *International Journal of Computer Science and Software Engineering* 4, 6 (2015), 141–155.

[33] E. Solomonik and L.V. Kale. 2010. Highly scalable parallel sorting. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–12.

[34] Hari Sundar, Dhairya Malhotra, and George Biros. 2013. HykSort: a new variant of hypercube quicksort on distributed memory architectures. In *International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10 - 14, 2013*. 293–302. DOI:http://dx.doi.org/10.1145/2464996.2465442

[35] Hari Sundar, Rahul Sampath, and George Biros. 2008. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing* 30, 5 (2008), 2675–2708. DOI:http://dx.doi.org/10.1137/070681727

[36] Hari Sundar, Rahul S. Sampath, Santi S. Adavani, Christos Davatzikos, and George Biros. 2007. Low-constant Parallel Algorithms for Finite Element Simulations Using Linear Octrees. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*. ACM, New York, NY, USA, Article 25, 12 pages. DOI: http://dx.doi.org/10.1145/1362622.1362656

[37] C. Walshaw and M. Cross. 2007. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In *Mesh Partitioning Techniques and Domain Decomposition Techniques*, F. Magoules (Ed.). Civil-Comp Ltd., 27–58. (Invited chapter).

[38] Albert-Jan N Yzelman and Rob H Bisseling. 2012. A cache-oblivious sparse matrix–vector multiplication scheme based on the Hilbert curve. In *Progress in Industrial Mathematics at ECMI 2010*. Springer, 627–633.

[39] Marco Zagha and Guy E Blelloch. 1991. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. ACM, 712–721.