Parallel Performance Optimizations on Unstructured Mesh-Based Simulations

Abhinav Sarje*, Sukhyun Song†, Douglas W. Jacobsen‡, Kevin A. Huck§
Jeffrey K. Hollingsworth†, Allen D. Malony§, Samuel W. Williams*, Leonid Oliker*
*Lawrence Berkeley National Laboratory {asarje,swwilliams,loliker}@lbl.gov
†University of Maryland {shsong,hollings}@cs.umd.edu
‡Los Alamos National Laboratory {douglasj}@lanl.gov
§University of Oregon {khuck,malony}@cs.uoregon.edu

***Abstract*—Distributed memory parallelization of unstructured meshes pose numerous challenges for attaining high performance and efficiency. In this paper, we address two of these primary challenges in the context of an ocean modeling code, MPAS-Ocean, which uses a mesh based on Voronoi tessellations: (1) load imbalance across processes, and (2) unstructured data access patterns, that inhibit intra- and inter-node performance. Load-balance and communication patterns are affected mainly by the grid partitioning strategy, while memory access patterns are impacted by the layout of data structures in the memory. Our work analyzes the load imbalance due to naive partitioning of the mesh, and develops methods to generate mesh partitioning with better load balance and reduced communication. Furthermore, we present methods that minimize both inter- and intra-node data movement and maximize data reuse. Our techniques include predictive ordering of data elements for higher cache efficiency, as well as communication reduction approaches. We present detailed performance data when running on thousands of cores using the Cray XC30 supercomputer and show that our optimization strategies can exceed the original performance by over 2×. Additionally, many of these solutions and can be broadly applied to a wide variety of unstructured grid-based computations.**

I. Introduction

Iteration-based computational simulations require an abstract representation of the real-world state. In most cases, the data is represented in a multidimensional collection of data points where some of the data points define spatial coordinates. In these applications, the spatial domain is decomposed into *cells* and the collection of cells forms a *mesh* or *grid*. In a distributed parallel implementation, these spatial coordinates often provide a convenient framework for decomposition of the data into independent, distributable blocks of cells as *partitions*.

Structured, or regular, meshes have regularly shaped cells and a regular connectivity. For example, 2D structured meshes are typically represented with rectilinear or curvilinear quadrangles which can often be easily decomposed into partitions. As expected, structured meshes are appropriate when they are sufficient for regular spatial domains. One problem with structured meshes is that because they are distorted in order to enclose certain shapes, such as the poles of cones and spheres, two or more vertex points share the same location, resulting in a *singularity*, where the tangent space is undefined.

On the other hand, *unstructured*, or irregular, meshes have arbitrary polygon cells and irregular connectivity. Often, unstructured meshes are represented with variable-sized triangles that intersect at high-radix nodes. As such, partitioning is often non-obvious and driven by complex algorithms. These meshes are useful in applications requiring either multiscale and/or variable resolution, and they map better to organic and curved surfaces (such as the Earth's surface). Furthermore, unlike structured meshes, unstructured meshes do not have singularities, or major distortions when mapped to 2D [1]. Applications where unstructured meshes are applicable include adaptive mesh refinement (other than block structured), multi-resolution domains, and finite element methods on an irregular domain.

Often domain decomposition for distributed memory computations require the use of *ghost* or *halo* cells from neighboring partitions when cells on the boundary of a partition require input from neighboring cells assigned to a different partition. Decomposition of unstructured meshes that are not fully connected results in irregular partition with potentially with high variability in the numbers of halo cells for each partition. This leads to load imbalance, increasing synchronization wait times and decreasing computational throughput. Furthermore, relatively large halo regions exacerbate this situation.

The on-node memory access patterns of structured and unstructured codes tend to be very different. Structured meshes are convenient in that their layout in memory is amenable to optimizations such as blocking, tiling, good cache reuse and vectorization because the cells are often represented as multidimentional rectangular arrays. Because of their regular shape and layout, they have a predicatable use pattern. In contrast, unstructured meshes are a performance challenge, as they are frequenly represented as connected graphs with non-contiguous memory layouts using pointers. Iterating over the "neighbors" of an unstructured mesh cell can result in poor cache reuse due to pointer chasing since the neighbors may not necessarily be local in memory.

In this paper, we address these two challenges put forth by unstructured mesh, load imbalance and unstructured data access patterns. The issue of load imbalance is discussed and addressed in Sections III and IV, and the unstructured data access issue is addressed in Section V. In addition, to

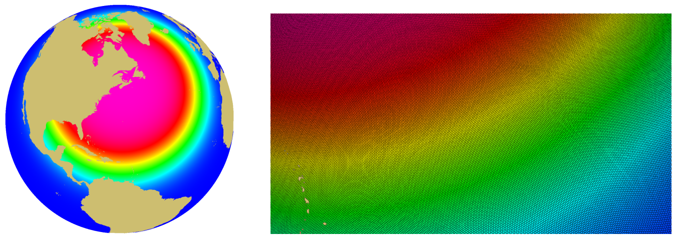


Fig. 1. (Left) Global variable resolution ocean mesh with refinement placed over the North Atlantic. Coloring is value of the generating density function. Pink is a value of 1, blue is a value of $(1/5)^4$. (Right) Detail of a region from the global variable resolution ocean mesh. Coloring is the same, with the actual mesh overlaid to show smoothness of mesh cells.

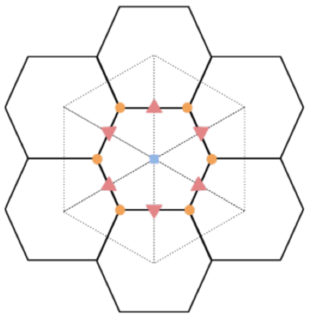


Fig. 2. Diagram showing MPAS-O horizontal staggering. Blue squares represent scalar data (mass, temperature, etc.), orange circles represent vorticity related quantities, red triangles represent vector quantities (velocities).

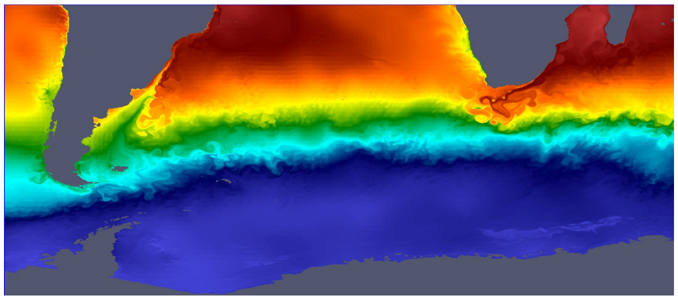


Fig. 3. Surface temperature from an MPAS-Ocean simulation after 90 simulated days. Using a variable resolution mesh that is 15km at the equator and 5km at the poles. Centered in the area of the Antarctic circumpolar current, in the south Atlantic.

reduce data movement, we present strategies for computation and communication avoidance in Section VI. We present our results and solutions in the context of an ocean modeling code, MPAS-Ocean [1], which we briefly describe in Section II-A.

II. Background and Related Work

We provide a description of the application at hand in the following. We also discuss the issues of load imbalance and unstructured data, and provide related work.

A. *Ocean Modeling with MPAS-O*

The Model for Prediction Across Scales (MPAS) is a modeling framework collaboratively developed between Los Alamos National Laboratory and the National Center for Atmospheric Research [1]. MPAS is written in C and Fortran, and is intended to provide developers with tools to enable rapid prototyping and development of climate model dynamical cores. The MPAS framework is built on top of unstructured mesh data structures that make heavy use of indirect addressing. The MPAS framework is publicly available[1] and currently contains four dynamical cores. These provide a shallow water model (MPAS-SW), and models of the Earths atmosphere (MPAS-A), land ice (MPAS-LI), and oceans (MPAS-O). This paper describes optimization efforts on unstructured meshes in the context of MPAS-O.

MPAS-O is a next generation global ocean model, built on top of spherical centroidal Voronoi tessellations (SCVT). The resultant meshes are composed of arbitrarily shaped polygons which bring unique challenges to load balancing and cache locality. In addition to containing arbitrary polygons, these meshes can be generated using a density function to place areas of static mesh refinement in areas of interest (such as the North Atlantic seen in Fig. 1). A major advantage of this type of mesh is that it provides smooth transition regions, also shown in Fig. 1, while allowing for drastically different resolutions in different regions of the mesh. It also eliminates the need for "hanging node" issues, that are common in regular structured variable resolution meshes, while still providing refinement around areas of interest. Building off of this unstructured mesh, MPAS-O enumerates its data on a staggered horizontal Arakawa C-Grid, visualized in Fig. 2. While the horizontal data structure is staggered and unstructured, MPAS-O has a vertical ("column") data structure that is regular and structured. This facilitates vectorization of column operations.

Global climate modeling is considered a grand challenge problem due to the spatial and temporal scales required to accurately simulate the phenomena. Temporal scales for climate models are on the order of centuries, while spatial scales are on the order of tens of kilometers. The requirements for performing global climate model simulations require models such as MPAS-O to simulate on large number of horizontal degrees of freedom of $O(10^6)$ to $O(10^7)$, with extremely short time steps. In addition, MPAS-O performs simulations using generally 100 vertical levels per horizontal degrees of freedom. Coupled together, these imply MPAS-O needs to run efficiently on high concurrency system to drive next-generation ocean analysis solutions. An example of an MPAS-O simulation result is shown in Fig. 3.

B. *Load Balance*

Load balancing across processing units in parallel applications is a widely researched topic due to its importance in gaining higher parallel efficiency. Load imbalance leads to underutilization of computational resources. Achieving a good load balance is particularly challenging when dealing with unstructured grids because their equal decomposition is not obvious [2], [3]. As unstructured grids are easily representable as graphs, many of the decomposition strategies for such

[1]http://mpas-dev.github.io

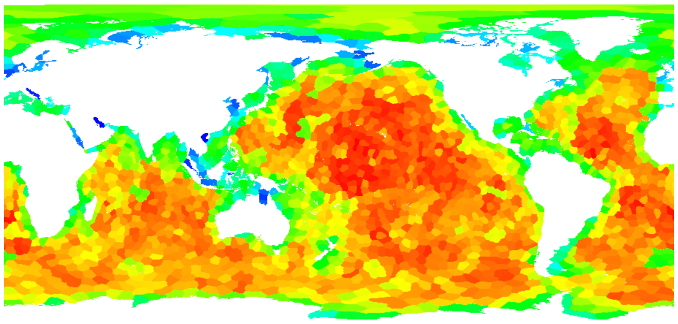


Fig. 4. An example visualization of the computational load imbalance of MPAS-Ocean using a grid with 1.8 million cells split into 2048 partitions, one per process. Bluer parts have low load and red parts have high load. In this example, the most utilized process performs approximately 3.26× as much work as the least utilized process, and approximately 30% of the mean execution time is spent in synchronization. Generally, computational load varies linearly with ocean depth and inversely with mesh spacing. Current partitioning scheme ignores the former.

grids are based on the graph partitioning algorithms. Although the graph partitioning problem is NP-complete, a number of heuristic-based partitioning algorithms exist [4]–[7].

The MPAS-O application uses a widely used graph partitioning tool, Metis [8] where the input is a graph representation of the unstructured grid. As will be discussed in Section III, using this approach directly is not quite effective, and causes significant imbalance across processes. A primary factor causing the imbalance is the use of multiple halo layers in this application, which has a major contribution to the communication as well as computation time. A visualization showing variability in computational loads across partitions in a sample execution of MPAS-O is shown in Fig. 4. Variable halo region cell counts and variability in the work per cell contribute to an unbalanced work distribution.

It is known that graphs do not model communication volume well. Hypergraphs, on the other hand, are able to model the communication volume accurately [9], [10]. Hence, in our work, we make use of hypergraph partitioning [10]–[13]. In Sections III and IV we will address this problem of load balance analyzing it in detail, and present our solutions.

C. Data Locality

Flops are free is the mantra of today's large-scale parallel systems. This refers to the observation that the primary expense in nearly all scientific applications today is data movement and not computations. This gap in the cost between computation and data movement is only going to grow larger in future. Hence, optimizing data movement (both volume and bandwidth) is essential in obtaining high performance. To affect this, the main strategies are (1) maximizing reuse of data once it has been loaded (reordering data traversal, and "communication-avoiding" algorithms) (2) maximizing bandwidth via streaming access patterns amenable to acceleration by hardware stream pre-fetchers.

To improve intra-node performance, it is essential that the memory hierarchy (caches) is used efficiently in data accesses. Data locality, both spatial and temporal, is an important factor in maximizing efficiency. When operating on unstructured grids, indirect and semi-random access to data presents a number of challenges to maximizing cache reuse and attaining high bandwidth. In this paper, we create data locality during the application execution by reordering the input mesh data. We use space-filling curves on the unstructured grid to reorder the cells to make sure that spatially nearby cells are stored nearby in the memory as well, which leads to effective use of the cache. Researchers have previously used various strategies for improving cache performance by introducing locality into structured grids [14], [15], including space filling curves [16] where although the authors consider unstructured data, they work with meshes with high-degree of regularity. We address this problem of intra-node data movement in Section V, discuss our solutions and present detailed performance analysis.

D. Computational Environment

The experiments described in this paper were executed on Edison, a Cray XC30 "Cascade" system at NERSC. Each Edison node has two 12-core "Ivy Bridge" sockets. Each socket is attached to 32 GB of DDR3-1600 DRAM via a memory controller providing about 44 GB/s of bandwidth. Nodes are connected via Cray's high-performance Aries (dragonfly) network. The dragonfly provides a scalable, low-latency network with dynamic routing to maximize bandwidth. In all the experiments presented in this paper, we run with 12 MPI processes per socket.

The data used in the experiments is a 15 km resolution world ocean grid. This grid has 1,848,103 cells and 5,509,887 edges. Each cell has between 3 and 40 levels of data, representing a normalized vertical depth of the ocean at that cell.

The MPAS framework utilizes unstructured meshes that can be created in a variety of ways but are typically Spherical Centroidal Voronoi Tessellations (SCVTs), as mentioned previously. Figure 2 shows the two cell meshes with overlapping spatial layouts: the *primary mesh* (Voronoi), and the *dual mesh* (Delaunay). Within each cell, edge, and vertex lie *scalar quantities* (blue squares), *vector quantities* (red triangles), and *vorticity quantities* (orange circles). After the mesh is partitioned, a typical partition has cells, vertices and edges explicitly assigned to that partition. During simulation initialization, each partition is also assigned a halo region that may contain multiple layers of cells from neighboring partitions. The number of required halo layers is algorithm dependent, based on the solver selected for the simulation.

The performance data presented in this paper has been collected using the TAU profiler tool [17].

III. Load Imbalance Analysis

MPAS-O uses the Metis tool to partition an input mesh represented as a graph into partitions, each of which is assigned to a different processor during simulations. The graph partitioner attempts to balance the number of cells among all the partitions, while minimizing the total number of edge cuts.

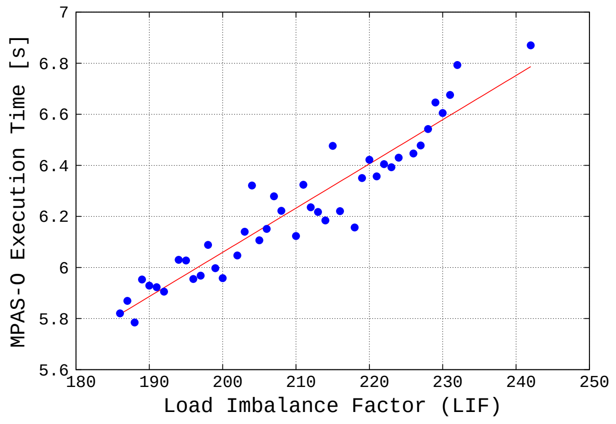


Fig. 5. The load imbalance factors, $\mathcal{I}$, for a partitioning has a strong positive correlation with the execution time. With different random number generator seed values, Metis generates partitioning with a wide range of $\mathcal{I}$ values and shows high performance variability.

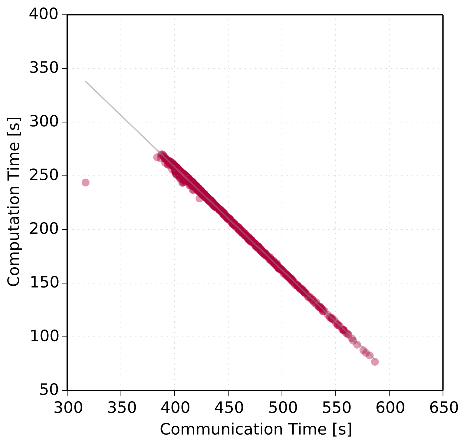


Fig. 6. The variation in the compute versus communication cost for processes during an example simulation using a partitioning obtained from Metis is shown. Each spot represents a different process. This simulation consists of 1536 processes, each owning a partition of the mesh. Communication time varies by about 42% and computation by nearly 75% showing a large imbalance. Detailed analysis shows that most of the communication time consists of wait time, where a process waits for others to finish their computations.

In solving the partial differential equations during simulations, certain operators require information in neighboring cells, which involves accessing additional data. These halo regions represent substantially larger domains than the simple edge cuts captured in the Metis partitioning. As remote processes continually update this data, frequent "halo-exchanges" are required. Thus, in order to improve MPAS-O performance, a partitioner must be cognizant of this data movement. In this section, we describe a compact set of experiments to explore and analyze the performance variability of numerous Metis partitioning solutions.

As a simple first step for understanding partition quality, we use the Metis partitioner to partition a smaller grid with 240 km resolution representing the world ocean into 24 partitions. A number of different partitioning are generated by using a different random number generator seed value for each. Since Metis is based on heuristics making use of randomization, each different seed results in a different partitioning. For each partition thus obtained, we compute a *load imbalance factor*, $\mathcal{I}$, which denotes the maximum of the number combined local and halo cells within a partition, such that minimizing $\mathcal{I}$ will effectively reduce the load imbalance.

Figure 5 shows the relationship between $\mathcal{I}$ and the overall MPAS-O execution time for a short 10-day simulation, with 43 different dispersed partitions obtained from Metis. Observe that with these partitions, the execution time correlates strongly with $\mathcal{I}$, and shows an overall performance difference of 17% between the minimum and maximum $\mathcal{I}$. It can be concluded from this that the performance can be improved by balancing the total number of local and halo cells per partition, as expected. Given the large performance variation due to different partitioning obtained using different seed values, a straightforward and naive strategy to obtain a better quality partitioning could be to run Metis a large number of times and select a partitioning with the minimum $\mathcal{I}$ value.

To demonstrate the computational and communication load imbalance, due to a given Metis partitioning, across the different processes during an execution, we show the variation in the computation and communication time for each of the process for a sample run in Fig. 6. It can be observed that in this example the compute time and communication time varies by nearly 75% and 42%, respectively, showing a large imbalance.

IV. Hypergraph-based Halo-aware Partitioner

Based on the above analysis it is clear that a different partitioning approach is required to achieve better load balancing, which would effectively balance the computation as well as the actual communication distribution. We thus turn to partitioning via hypergraphs using Patoh [18], which promises to model the communication overhead more accurately.

As previously described, the communication overhead between processors is due to exchange of data in the "halo" regions, and due to the unstructured nature of grids, each partition has variable number of halo cells dependent on the partitioning. The effect of halo cells is magnified when there are multiple halos required, as is the case in MPAS-O that typically requires three halos layers per partition. Since graph and hypergraph partitioning algorithms do not take these halo cells into account, a different partitioning strategy is necessary for a better balance. In order to develop a *halo-aware* partitioning scheme, we first construct a performance model for a given partitioning, and then use this model to design our partitioning algorithm.

A. Computation and Communication Cost Modeling

The computational cost for a processor includes the cost due to computations on the local cells and on the halo cells. The cost may be different for a local cell and a halo cell. For p total partitions, let a represent the ratio between computation

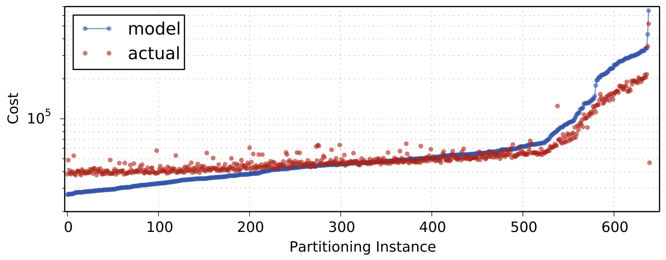


Fig. 7. Cost prediction from the performance model compared with actual runtime cost with respect to a total of about 700 different partitioning obtained by using Patoh directly with different configurations.

performed on a halo and local cells. Given a partition k, we model the computational cost, C_α as:

$$C_\alpha = \frac{1}{F(p)} \left(\sum_{i}^{i \in N_k} w_i + a \sum_{i}^{i \in H_k} w_i \right) \tag{1}$$

where N_k is the set of local cell for partition k, H_k is the set of halo cells for partition k, and w_i is weight of a cell i. $F(p)$ is a function of p, since the cost depends on the number of partitions as well.

The communication among processors can be viewed as "halo-exchanges", where the communication volume depends on the number of halo cells for each partition, and the number of neighbors each processor needs to communicate with. We model the communication cost of a partition k as:

$$C_\beta = \frac{1}{F(p)} \frac{h_k}{b_k} + \left(\max_i(c_i) - c_k \right) \tag{2}$$

where $h_k = |H_k|$, b_k = number of neighbors of partition k, c_k is the total computational cost of partition k. Hence, the second term represents the wait time for processor owning partition k.

We perform extensive experiments with the actual cost of representative simulations with varying concurrency and input meshes, and use the obtained performance data to perform a least-squares line fit in order to find the computation-communication ratio, and the value of $F(p)$. From the data collected from these experiments, we obtain $a = 0.7$ and $F(p) = \log(4p)$. Figure 7, confirms that our predicted performance modeling results accurately estimate the actual application runtime for any given mesh partitioning.

B. Halo-aware Partitioning

To model multi-layered halos, we could naively use BFS or DFS starting from each cell and identify all the k distance neighbors to construct k-layered halos. However, given that graphs can be represented as sparse matrices, a more accurate approach is to compute the k-th power of a matrix, through sparse matrix-matrix multiplication (SpMM), which identifies all nodes in the corresponding graph that are at a distance k and connects them with an edge. Therefore, given A as the sparse matrix representation of the input mesh, computing A^k determines the k halo layer cells. We use the CombBLAS package to perform the SpMM operations [19]. Aggregating the edges from $A, A^2, ..., A^k$ identifies all the halo cells for any given partitioning.

However, because the halo cells cannot be identified until the partitioning has been performed, their cost cannot be added for balancing during the partitioning. Therefore, our new partitioning strategy follows an iterative approach. The strategy is designed as a Monte-Carlo based partitioning scheme that iteratively refines the partitioning using total cost computed for previous iteration's partitioning. Each iteration, thus, includes a call to the hypergraph partitioning tool Patoh. This scheme proceeds as follows:

1) Construct sparse matrix A representing the input mesh.
2) Compute $A^2, \cdots, A^k$ to identify potential halo cells for each cell.
3) Construct a hypergraph representation $\mathcal{H}_0$ of the sparse matrix $A_{1\cdots k}$, which is an aggregation of $A, A^2, \cdots, A^k$.
4) Iterate over the following until convergence:
   a) Compute a partitioning P_i of the hypergraph $\mathcal{H}_i$.
   b) Construct halos for partitioning P_i.
   c) With P_i, use the performance model to calculate total cost prediction C_k for each partition k and assign corresponding weights to the cells (distributing the cost of halo cells among the partition cells), and hence compute the partition weights W_k.
   d) Calculate imbalance factor, to be used as "fitness" value, $f_i = \left(1 - \frac{\min_k(W_k)}{\max_k(W_k)}\right)$.
   e) If the new fitness f_i is less than f_{i-1}, accept this partitioning P_i.
   f) Otherwise, calculate the *Metropolis-Hastings* probability $m = \min\left(1, e^{\frac{f_{(i-1)} - f_i}{T}}\right)$, and accept the partitioning with this probability.
   g) If the partitioning is accepted, then update the hypergraph with the new cell weights constructing $\mathcal{H}_{(i+1)}$, and continue with the next iteration unless convergence was reached.
   h) Otherwise, reject the partitioning by setting $P_i = P_{i-1}$ and $f_i = f_{i-1}$, and continue to the next iteration.
5) Output the last accepted partitioning as the result.

The above procedure ensures that the cost due to halos are taken into account by assigning weights to the cells and refining the partitioning. Experimental results show that the convergence is reached quickly, generally within 10 iterations. The effect of the cost/weight distribution and the resulting improved partitioning can be seen in the visualization of Figure 8, showing 384 partitions.

C. Variable Cell Weights

In general a mesh may have variable cell weights. For MPAS-O the cells in the input mesh have variable depths, representing the ocean depth at the corresponding location. This introduces another opportunity for optimization, since the computational cost of a cell is proportional to the number of

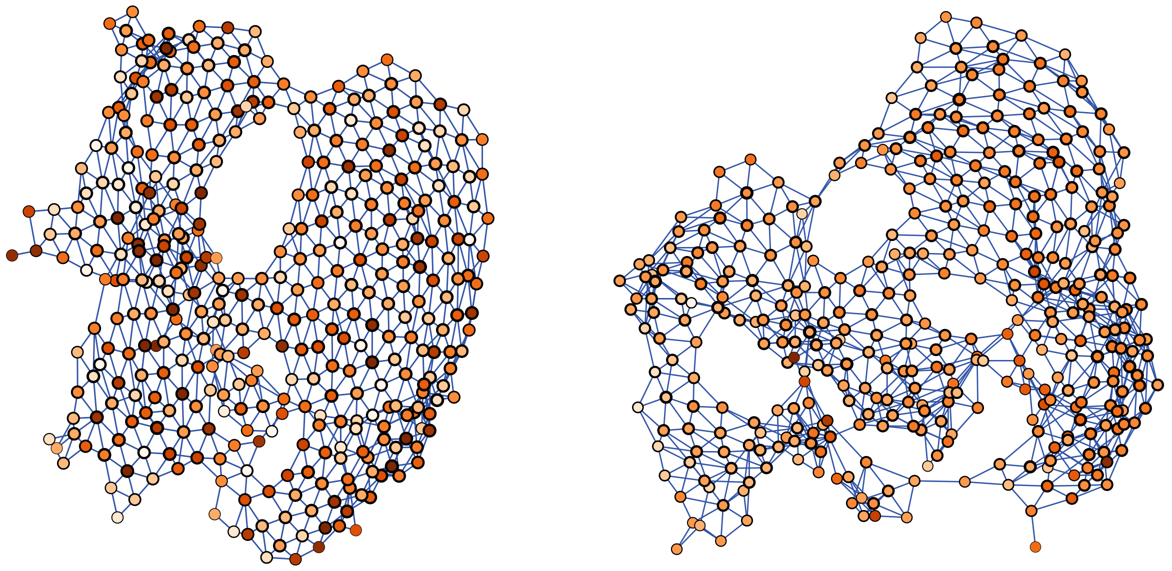


Fig. 8. Comparing the weights of each partition visually. The graph represents a partitioning, with nodes as individual partitions with edges between adjacent partitions. Weight of a partition is depicted in its color – darker means higher weight and lighter means lower weight. (Left) The original graph partitioning obtained from Metis. (Right) A new partitioning obtained from our halo-aware hypergraph partitioning strategy. The variation in partition weights can be clearly seen to be much higher in original compared to the new partitioning.

depth layers it defines (varying between 3–40). To take this into account, variable weights corresponding to the cell depth are assigned to each cell while calculating the cost using the performance model. The next subsection presents the impact of our depth and halo-aware partitioning strategies compared with assigning the original partitioning.

D. Performance Results

We perform experiments on the Edison system (described in Section II-D), using a 15 km resolution mesh to simulate 10 days, with the following four partitioning strategies:

1) The original graph partitioning obtained using Metis.
2) Direct hypergraph partitioning obtained using Patoh.
3) Partitioning obtained from our halo-aware partitioner.
4) Partitioning obtained from our depth and halo-aware partitioner.

Each partitioning used in the following experiments was selected from a sample size of 10 obtained from the corresponding partitioning scheme, on the basis of best performance.

The performance as communication time and computation time of each run is shown in Figures 9 and 10, respectively. These plots capture the maximum to minimum range of these times. Observe that the communication time drops significantly for the partitions obtained from our new partitioners, with a small increase in the computation time. This is because the new partitioner attempt to combine the computation and communication cost, and better overall cost may be achieved by increasing the maximum compute time and decreasing the maximum communication time. It should be noted that these plots do not show the correspondence between the minimum and maximum of computation and communication time for each process, and total time taken is not simply the sum of the two.

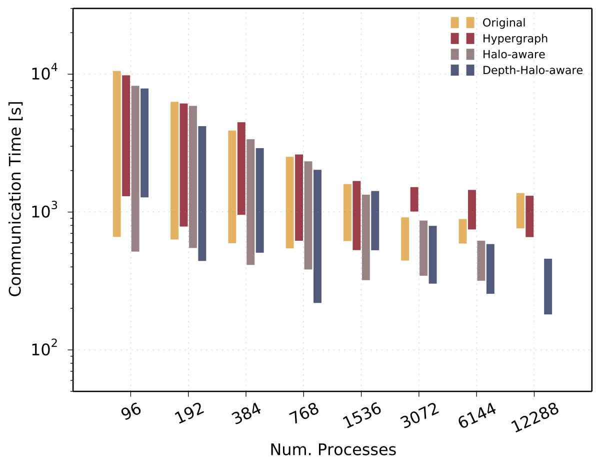


Fig. 9. The minimum to maximum range of time involved in communication related tasks among all processes in each run is shown. Comparison of the four types of partitioning is depicted for varying number of processes. The number of partitions is equal to number of processors, and each partition is assigned to a different processor.

The total execution time taken by each run is shown in Figure 11, where it can be seen that at 12,288 cores the performance improvement of our scheme is more than 2× over the base graph partitioning (as well as direct hypergraph partitioning). Lower concurrencies also show performance improvements, although less dramatic. Note that due to the significant reduction in communication costs, our partitioning scheme improves performance as well as scalability.

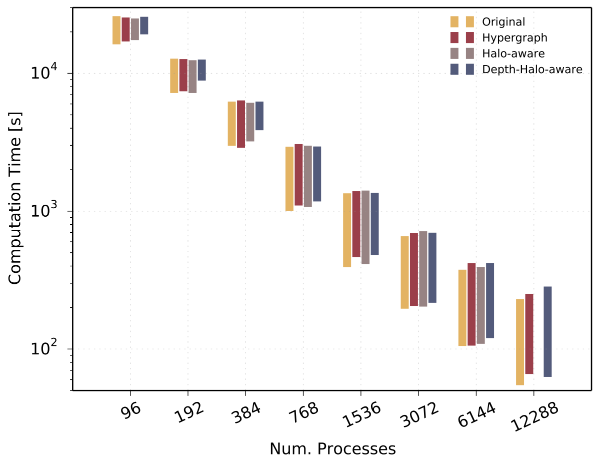


Fig. 10. The minimum to maximum range of time spent in computations among all processes in each run is shown. Comparison of the four types of partitioning is depicted for varying number of processes. The number of partitions is equal to number of processors, and each partition is assigned to a different processor.

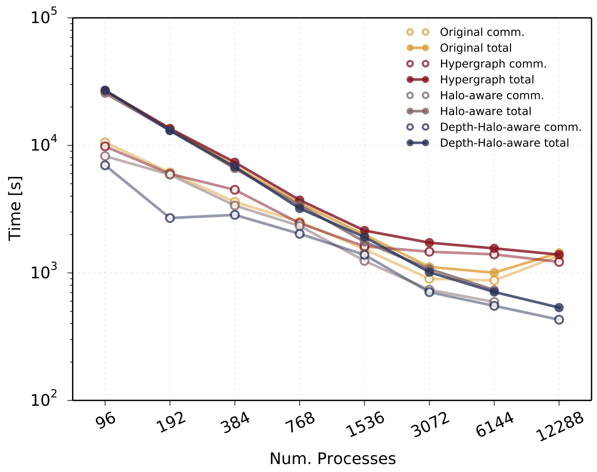


Fig. 11. The total application execution times with varying number of processes are shown for each of the four types of partitioning. The number of partitions is equal to number of processors, and each partition is assigned to a different processor. Corresponding communication time is also shown in lighter color for reference.

V. Ordering Unstructured Data

An unstructured grid implicitly means that the data layout is also unstructured, which causes the involved data structures to be mapped to the memory in no particular order. There is almost no locality, leading to inefficient use of the memory hierarchies. In our case, the meshes being considered have no regularity, and no assumptions regarding the data ordering can be made. Since maintaining locality is essential for high performance in presence of memory hierarchies, we need to create spatial data locality by re-arranging data so that the data with temporal locality reside nearby in the memory as much as possible. Currently, since the MPAS-O mesh can be directly represented as a sparse matrix, to create locality, its generation process follows a cell ordering called the *reverse Cuthill-McKee ordering* [20], which is an algorithm for permuting a symmetric sparse matrix to convert it into a band matrix with a small bandwidth. We will show below that even though it follows some locality pattern, its performance is nearly as bad as a complete random ordering.

A. Space Filling Curves

Space Filling Curves (SFCs) [21], [22] are generally used to map a higher dimensional data on to a one-dimensional line. A number of SFCs have been developed and have proven to be widely applicable for solving scientific problems as well as for improving performance of applications. In our case, SFCs are an obvious choice to explore for the creation of data locality. But due to the lack of any regularity in the underlying grid, SFCs cannot be directly applied since SFCs are defined for regular and/or repeating grid patterns. But fortunately, the SCVT mesh under consideration exists in 3-dimensional space. Hence, we can overlay a structured grid on top of the unstructured mesh, for the purposes of reordering with SFCs.

Given an unstructured mesh defined in a 3-dimensional space, we represent each cell as a single point at the position of its centroid. Then, we define an octree based structured rectangular grid covering all these points. This grid is decomposed recursively such that each point resides in a separate leaf node. Once we have this representation, we simply order these octree nodes according to a SFC on this structure. This leads to an ordering of the points which represent the cells of the unstructured mesh. We then simply reorder these cells in the data file according to this constructed ordering.

Due to the definition of our mesh on a spherical surface, and with discontinuities, the application of the above strategy to the complete mesh at once causes significant number of "jumps" between cells, along the SFCs, which are not spatially close, but appear next to each other in the ordering. Hence, instead of reordering the whole mesh, we apply the ordering to each partition separately once partitioning has been done. This minimizes the number of such jumps and allows for better data locality.

We consider several SFCs in order to evaluate their affect on simulation performance, and here we present two such SFC based orderings: Hilbert curve, and Morton ordering (or, Z-SFC). Examples of the ordering of the mesh cells produced by these curves is shown in Fig. 12. Also shown in the same figure are our two base cases: the original reverse Cuthill-McKee ordering, and a randomized ordering. In addition to ordering the cells, we follow the same scheme to reorder the other data elements in the mesh: the vertices and the edges (see Fig. 2).

B. Performance Impact

We reorder the mesh data elements according to the above described scheme and perform experiments to analyze the performance improvements they create. In Fig. 13, the results are summarized as the total application execution time taken using each of the following four mesh orderings:

1) Completely randomized ordering, for reference.

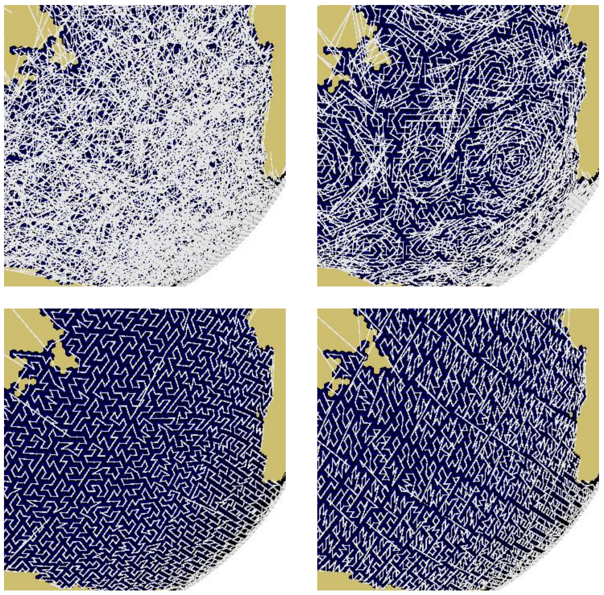


Fig. 12. A part of the mesh covering Earth's oceans is shown for each of the four cell orderings: random (top-left), original (top-right), hilbert (bottom-left) and morton (bottom-right). The ordering of the cells is shown using white lines connecting cells.

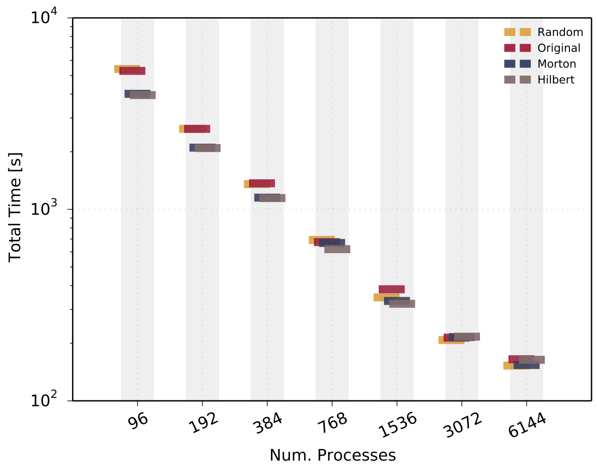


Fig. 13. Scaling of the total application execution time for the four mesh data orderings is shown. Although at lower concurrency, the performance improvement is obvious for SFC-based ordering, their benefits diminish at higher concurrency due to small sized partitions.

2) Original Cuthill-McKee ordering, as the base case.
3) Morton SFC ordering.
4) Hilbert SFC ordering.

All these experiments have been performed using the depth and halo-aware partitioning obtained from previous section. Hence, the observed benefits are on top of the previous improvements. The benefit of introducing data locality into the mesh by using the two SFCs are clearly observed with smaller number of partitions, and the performance gain gradually diminishes. Since this is strong scaling, the observed behavior is expected because as the partition size grows smaller, more of the mesh data can fit into the caches, which makes their ordering not as important. Nevertheless, the benefits will be more profound in high resolution meshes as they scale to even larger number of processors.

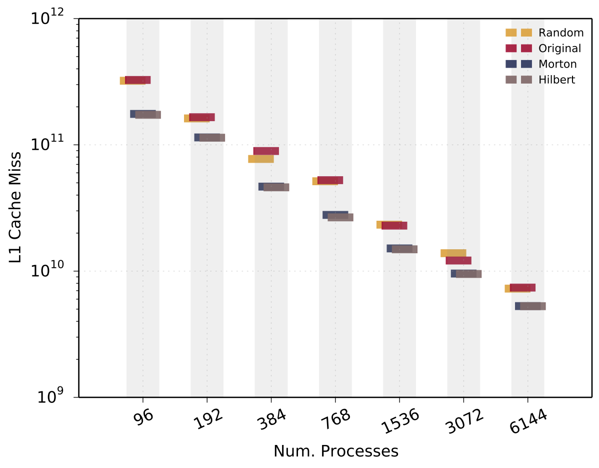


Fig. 14. Total number of L1-cache misses during the application execution with the four mesh data orderings is shown. The savings in data movement due to SFC-based ordering can be seen.

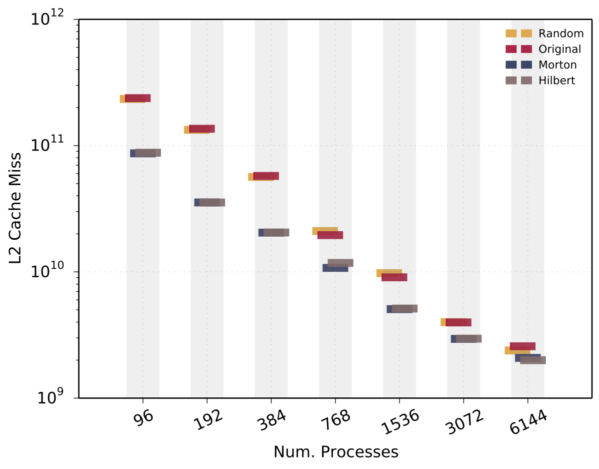


Fig. 15. Total number of L2-cache misses during the application execution with the four mesh data orderings is shown. The savings in data movement due to SFC-based ordering can be seen, being more significant at lower concurrencies as the data fits into the cache.

Further, to analyze how the cache behavior is affected by the reordering, we collect cache statistics using hardware counters. The total number of cache misses for L1, L2, L3 caches and TLB, during the execution of the applications are shown in Fig. 14 to Fig. 17, respectively. These show that ordering the data is definitely beneficial for improving cache performance. Apart from reducing the execution time with reduced cache misses, the minimization of data movement also leads to lower power consumption.

VI. Computation and Communication Avoidance

We now explore the application of computation and communication avoidance techniques to the MPAS-O model. When

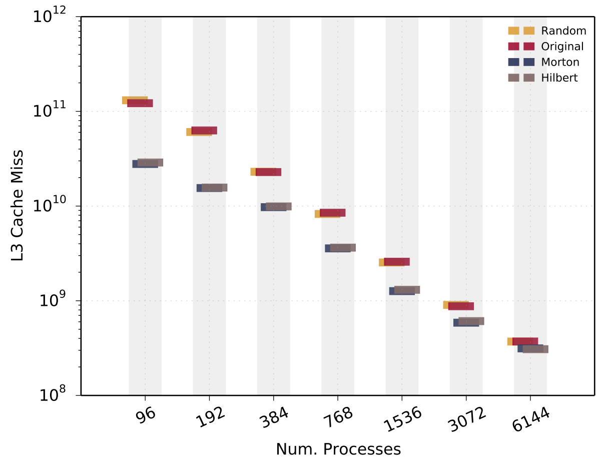


Fig. 16. Total number of L3-cache misses during the application execution with the four mesh data orderings is shown. The savings in data movement due to SFC-based ordering can be seen, being more significant at lower concurrencies as the data completely fits into the cache.

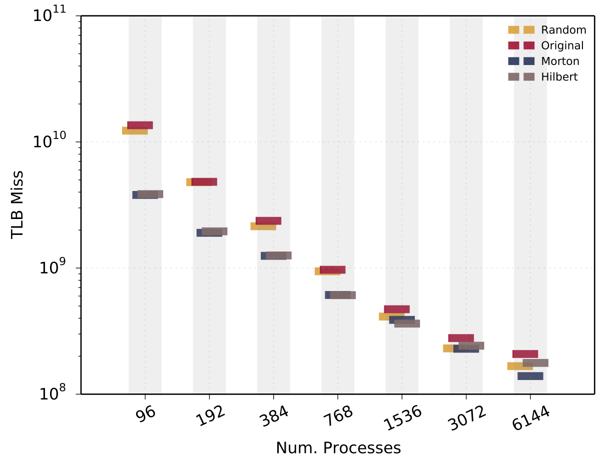
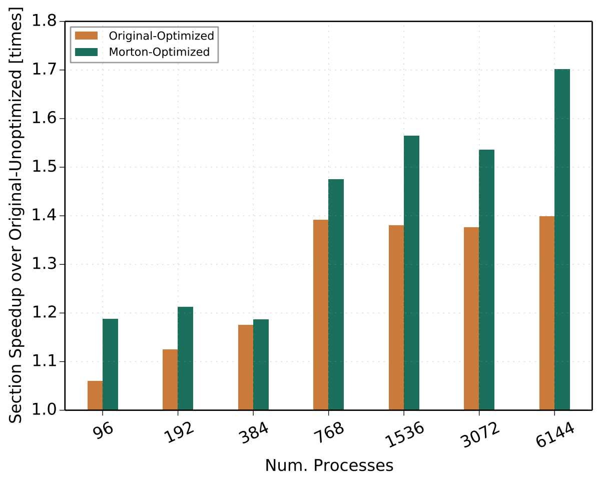


Fig. 17. Total number of TLB misses during the application execution with the four mesh data orderings is shown. The savings in data movement due to SFC-based ordering can be seen.



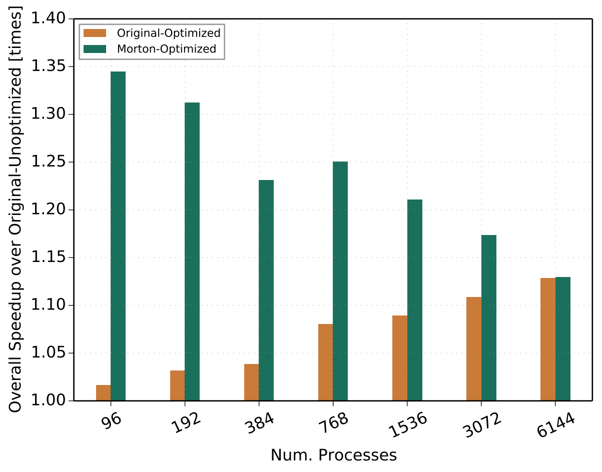


Fig. 18. Speedups achieved by the computation and communication avoiding technique for the optimized code segment (top) and the overall application execution time (bottom) are shown. The speedup obtained with the Morton ordered mesh and the optimized code is up to 1.34×.

decomposed, each processor is required to correctly compute on a subset of the total domain. Some of the operators used to solve MPAS-O's equations require information from neighboring elements (i.e. cells, edges, and vertices) which might be stored in the regions halo layers. During the course of a time step, MPAS-O will compute over all halo layers without requiring any communication. However, prior to advancing to a new time step, MPAS-O requires an update of the halo layers to ensure data contained in the halo is identical to data for those same elements on other processors. We implement a technique to modify the general computation and communication pattern for one particular segment of the MPAS-O code. This technique eliminates redundant computation and communication with k layered halos. by executing the following during each time step:

1) For k steps, where in i-th step, compute information on local cells plus $(k - i - 1)$ halo layers.
2) Perform a halo-exchange operation with all k layers.

Each step corrupts the data in the $(k - i + 1)$-th halo layer, hence, a halo-exchange is performed only after all halo layers have corrupted data. This scheme avoids communication after each computation step, and performs one halo-exchange per k steps. Apart from avoiding communication, this scheme also avoids redundant computations on the halo layers which have corrupted information.

The results of the redundancy avoidance optimizations obtained from experiments on Edison are shown in Fig, 18. Here, "original" refers to the original 15 km resolution mesh, and "Morton" represents our SFC reordered mesh as described in the previous section. We compare the performance with optimizations described above using these two meshes, against the original code (unoptimized). The speedups observed for these two cases are shown in Fig. 18.

Observe that the achieved overall application speedup due to this optimization, over the baseline code is up to 1.13×, while the speedup observed for just the optimized code segment is

1.4×. Additionally, the combined speedup from Morton ordering in conjunction with the redundancy avoidance optimization is up to 1.34× for the overall execution time and 1.7× for the optimized code segment.

Since this technique removes two-thirds of communication in this code segment, the achieved speedup improves with increasing concurrency for strong scaling.

VII. Conclusions

In this work we address two of the primary challenges of improving load balance and data locality, when faced with computations over unstructured meshes. We demonstrated that distributing a work load across processors with an unstructured mesh partitioning, it is necessary to consider the computation and communication costs due to any halos used in order to reduce overall communication and gain higher performance. We therefore devised a new hypergraph-based depth and halo-aware partitioning strategy, which allowed significant improvement for MPAS-O at scaling, attaining more than 2× speedup at scale compared with the original partitioning strategy.

We additionally applied an SFC-based ordering to the unstructured mesh by mapping it to a 3D regular domain, showing that ordering is essential to reduce data movement when the data size is significantly larger than the system cache size. Finally, we explored the application of communication avoidance techniques to a segment of MPAS-O's internal workload, showing how portions of the model can benefit from this technique.

Acknowledgments

The authors thank Aydin Buluc and Umit Catalyurek for helpful discussions on graph partitioning. Authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05CH11231. Authors from the University of Oregon are supported by DOE SciDAC grant DE-SC0006723. Douglas Jacobsen is supported by the US DOE Office of Science, Biological and Environmental Research program. We used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] T. Ringler, M. Petersen, R. L. Higdon, D. Jacobsen, P. W. Jones, and M. Maltrud, "A multi-resolution approach to global ocean modeling," *Ocean Modeling*, vol. 69, no. C, pp. 211–232, Sep. 2013.

[2] Y. F. Hu and R. J. Blake, "Load balancing for unstructured mesh applications," *Parallel and Distributed Computing Practices*, 1999.

[3] M. Berzins, "A new metric for dynamic load balancing," *Applied Mathematical Modelling*, 2000.

[4] C. Walshaw, M. Cross, and M. G. Everett, "Dynamic mesh partitioning: A unified optimisation and load-balancing algorithm," 1995.

[5] K. Schloegel, G. Karypis, and V. Kumar, "Parallel static and dynamic multi-constraint graph partitioning," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 3, pp. 219–240, 2002.

[6] I. Moulitsas and G. Karypis, "Architecture Aware Partitioning Algorithms," in *ICA3PP '08: Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*. Springer-Verlag, Jun. 2008.

[7] J. M. Dennis, "Inverse space-filling curve partitioning of a global ocean model," in *International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007, pp. 1–10.

[8] G. Karypis, *METIS: A Software Package for Partitioning Unstructured-Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*, Mar. 2013.

[9] Ü. V. Catalyürek, "Hypergraph models for sparse matrix partitioning and reordering," Ph.D. dissertation, 1999.

[10] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Catalyürek, "Parallel hypergraph partitioning for scientific computing," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, p. 10 pp.

[11] Ü. V. Catalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 10, no. 7, pp. 673–693, 1999.

[12] B. Uçar and C. Aykanat, "Revisiting hypergraph models for sparse matrix partitioning," *SIAM review*, vol. 49, no. 4, pp. 595–603, 2007.

[13] N. Selvakkumaran and G. Karypis, "Multi-Objective Hypergraph Partitioning Algorithms for Cut and Maximum Subdomain Degree Minimization," *IEEE Transactions on Computer Aided Design*, pp. 1–14, Apr. 2005.

[14] M. A. Bender, B. C. Kuszmaul, S.-H. Teng, and K. Wang, "Optimal Cache-Oblivious Mesh Layouts," *Theory of Computing Systems*, vol. 48, no. 2, pp. 269–296, Feb. 2011.

[15] A. Coutinho and M. Martins, "Performance comparison of data-reordering algorithms for sparse matrix–vector multiplication in edge-based unstructured grid computations," *International Journal . . .*, 2006.

[16] H. T. Vo, C. T. Silva, L. F. Scheidegger, and V. Pascucci, "Simple and Efficient Mesh Layout with Space-Filling Curves," *Journal of Graphics Tools*, vol. 16, no. 1, pp. 25–39, 2012.

[17] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, May 2006.

[18] U. Catalyurek and C. Aykanat, *PaToH: Partitioning Tool for Hypergraphs*, March 2011.

[19] A. Buluc and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *International Journal of High Performance Computing Applications*, 2011.

[20] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69, 1969, pp. 157–172.

[21] M. Bader, *Space-Filling Curves*, ser. An Introduction With Applications in Scientific Computing. Springer, Oct. 2012.

[22] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the Hilbert space-filling curve," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 1, pp. 124–141, 2001.