IMPROVING UNSTRUCTURED MESH PARTITIONS FOR MULTIPLE CRITERIA USING MESH ADJACENCIES*

CAMERON W. SMITH†, MICHEL RASQUIN§¶, DAN IBANEZ†, KENNETH E. JANSEN§, AND MARK S. SHEPHARD†

Abstract. The scalability of unstructured mesh based applications depends on partitioning methods that quickly balance the computational work while reducing communication costs. Zhou et al. demonstrated the combination of (hyper)graph methods with vertex and element partition improvement for PHASTA computational fluid dynamics scaling to hundreds of thousands of processes. Our work generalizes partition improvement to support balancing combinations of all the mesh entity dimensions (vertices, edges, faces, regions) in partitions with imbalances exceeding 70%. Improvement results are presented for multiple entity dimensions on up to one million processes on meshes with over 12 billion tetrahedral elements.

Key words. partition improvement, graph/hypergraph, unstructured mesh, dynamic load balancing

AMS subject classifications. 65Y05, 68W10

1. Introduction.

Parallel simulation-based engineering workflows using unstructured meshes require adaptive methods to ensure reliability and efficiency [56]. Starting with a problem specification on a geometric model [40, 55], an effective workflow automatically executes parallel mesh generation [63], analysis, and analysis-based mesh [11, 45] and/or model [43] adaptation. The analyze-adapt cycle is repeated until a desired level of solution accuracy is reached. Between each step in the cycle is an opportunity to improve scalability and efficiency through dynamic partitioning.

Current dynamic load balancing methods do not effectively reduce imbalances to the levels needed by applications capable of strong scaling to the full size of leadership class petascale systems. This paper presents a scalable approach that quickly reaches the required imbalance levels for multiple criteria by pairing ParMA, Partitioning using Mesh Adjacencies, with current partitioning methods. Section 2 introduces the dynamic partitioning problem then reviews (hyper)graph, geometric recursive sectioning, and diffusive partitioning methods. Section 3 provides our contributions, describes how they satisfy the dynamic partitioning problem, and then describes the partition improvement procedures. Section 4 begins with a comparison of ParMA and its predecessor, LIIPBMod. Next, we present a ParMA feature comparison test and multi-criteria partitioning results on meshes with over 12 billion elements running on over one million cores. Section 4 closes with a discussion of scaling improvement in a

*Submitted to the journal's Software and High-Performance Computing section DATE; accepted for publication (in revised form) DATE, published electronically DATE. This research was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under award DE-SC00066117 (FASTMath SciDAC Institute). We gratefully acknowledge the use of the resources of the Rensselaer Center for Computational Innovations and the Leadership Computing Facility at Argonne National Laboratory. Before joining Cenaero, M. Rasquin was funded for a significant part of this work by the ALCF Early Science Program which is also acknowledged.

†SCOREC, Rensselaer Polytechnic Institute, 110 8th St., Troy, NY 12180 (smithc11@rpi.edu, ibaned@rpi.edu, shephard@rpi.edu)

‡University of Colorado Boulder, 1111 Engineering Dr, ECOT 126 Boulder, CO 80309 (michel.rasquin@colorado.edu, kenneth.jansen@colorado.edu)

§Cenaero, Rue des Frères Wright 29, Bâtiment Eole, 6041 Gosselies Belgium (michel.rasquin@cenaero.be)

CFD analysis running on over a half-million cores. Section 5 concludes the paper.

2. Unstructured Mesh Partitioning. The dynamic partitioning problem seeks to quickly improve the load balance and reduce communication costs of an existing partition that is reasonably distributed; such as those generated by (hyper)graph and geometric partitioning tools. Hendrickson and Devine [25] define the requirements of dynamic partitioning as: (1) balance the computational work, (2) reduce the inter-processor communication costs, (3) modify the partition incrementally, (4) output the new communication pattern, (5) execute on parallel systems quickly, (6) consume small amounts of memory, and (7) provide an easy to use functional interface. For unstructured meshes these requirements are mostly satisfied by multi-level (hyper)graph and recursive sectioning methods [52]. Multi-level (hyper)graph methods are limited in scalability; memory requirements limit their effective usage on more than several thousand processors [24]. Recursive sectioning methods are limited in quality; they have lower memory and time requirements at the expense of increased inter-part surface area. Additionally, these methods can only balance one dimension of mesh entity. This approach can result in a less-than optimal balance of the other entity dimensions as process counts increase. The balance of the other dimensions can be improved, but not fixed, with carefully defined weights in the multi-constraint partitioning options provided by Zoltan's recursive coordinate bisection implementation and by the multi-level (hyper)graph methods [1, 34, 51]. Below, we review the graph, geometric sectioning, and diffusive partitioning approaches in more detail.

2.1. (Hyper)Graph Partitioning. Graph-based partitioning methods define an assignment of weighted graph nodes to k parts such that each part has the same total weight and the inter-part communication costs are minimized. A graph is constructed from an unstructured mesh by selecting one dimension of mesh entity (i.e., vertices, edges, faces, or regions) to define graph nodes, and one mesh adjacency between the selected entity dimension to define graph edges. At a higher level, the goal of this selection is to represent a work unit with the graph node and an information dependency between two work units by a graph edge. 3D element-based finite element and finite volume codes typically select mesh regions for graph nodes and mesh faces shared by elements for graph edges. This selection results in the unique assignment of mesh regions to parts, which enables efficient local execution of element-level computations [28].

Parallel, multi-level, graph-based partitioning methods produce high quality partitions with tens of thousands of parts in a fraction of the time needed by most analysis procedures [7, 35, 37, 51]. One approach to generalize these methods to represent more complex information dependencies uses hypergraphs. A hypergraph is defined as a set of weighted nodes and hyperedges. Hyperedges differ from graph edges in that they represent dependencies between multiple graph nodes and, in doing so, have the ability to better model the communication costs of an application [8, 9]. As with graph-based partitioning, the goal of hypergraph partitioning is to balance the node weight across the k parts while minimizing a hyperedge-based objective function. Boman and Devine propose constructing the hypergraph from an unstructured mesh by creating one hypergraph node for each mesh region (in 3D), as is done in the graph-based construction, and a hyperedge connecting the mesh regions bounded by each mesh vertex. This richer representation improves the modeling of communication costs, but results in algorithms that are more compute and memory intensive relative to graph-based methods.

2.2. Geometric Partitioning. Geometric methods represent information via spatial coordinates, and relations via distance; the closer two pieces of information are the stronger their relation. The exclusive use of coordinate information significantly reduces the memory requirements of these methods relative to (hyper)graph methods that rely on topological relations [24]. Along with the lower memory cost, the spatial sorting procedures used by geometric methods are also computationally cheaper than the topological traversals needed by graph methods. The lower computational and memory usage costs come at the expense of significant increases in inter-part communications [48]. For applications that require frequent balancing though, the resulting communication overheads may be offset by the time saved computing the partition [24].

Geometric recursive sectioning methods can quickly compute well-balanced partitions for a single entity dimension [6, 15, 47, 62, 68]. Recursive coordinate (RCB) [4] and inertial bisection (RIB) [57, 62, 68] methods recursively cut the parent domain; RCB along a coordinate axis and RIB perpendicular to the parent domain's principal direction. Multi-sectioning techniques [15, 47] can be considered extensions of the recursive coordinate bisection methods as they define cuts along coordinate axis, but do so with multiple parallel cut planes at each recursion.

Partitioning methods using space-filling curves (SFC) produce partitions of similar quality to RCB and RIB. For 3D unstructured meshes Hilbert [58] and Morton curves have been used effectively by the Zoltan [17] and SPartA [24] packages, respectively. Given the simplicity of SFC partitioning methods (encoding, sorting, then splitting) a high degree of on-node and inter-node concurrency is possible. For example, a constant time Hilbert curve encoding procedure (spatial coordinates to curve position) [58] and its subsequent sorting has been demonstrated on shared-memory devices using a data-parallel implementation [30] and a two-collective splitting approach is used by SParTA. As an added benefit, sorting provides a cache efficient layout of the mesh entities for subsequent mesh-based operations that benefit from topological locality [23, 71].

2.3. Diffusive Partitioning. Diffusive partitioning methods efficiently improve an existing partition by transferring load between neighboring parts. Load transfer can be coordinated globally or locally. Global load transfer selects elements to minimize either the total weight of transferred elements, or the maximum weight transferred in to or out from a part [26, 27, 38, 41, 49, 50, 64]. Alternatively, local load transfer iteratively moves elements from heavily loaded to less loaded parts [61, 14, 49, 67]. This approach can have significantly lower overall computational costs if the total amount of transferred load is controlled. Control is typically exerted through greedy heuristics. These heuristics first determine the amount of load to transfer between neighboring parts, and then select elements to satisfy the transfer requirement. Fiduccia [19] and Kernighan [36] proposed selecting elements based on the subsequent part quality improvement. For partitioning complex graphs with up to one trillion edges [59, 60] these heuristics have proven successful as part of a label propagation based approach. Likewise, a greedy improvement heuristic is applied to reduce the communication cost of parallel sparse matrix-vector multiplication [5]. In Zhou's work on unstructured meshes a similar heuristic is shown to be highly scalable given a distributed mesh representation [70, 72].

3. Partitioning Using Mesh Adjacencies. Zhou's 2010 work [70] defines the LIIPBMod algorithm for reducing vertex imbalance and the number of vertices on part boundaries while indirectly trying to limit the increase of element imbalance. In

2012, Zhou [72] executes a strong scaling study of a massively parallel computational fluid dynamics application using partitions created with (hyper)graph partitioners and LIIPBMod. Our work, ParMA, defines new algorithms for balancing all entity dimensions (vertices, edges, faces and regions), with weights, while reducing the number of vertices on the part boundaries, the number of disconnected components, and the average number of neighboring parts. ParMA developments were guided by Zhou's work for vertex balancing.

Our work, relative to Zhou's, demonstrates multi-entity balancing on up to 3.5 times more parts, 1Mi, with up to two times smaller parts, 1100 elements. Like Zhou, we focus on balancing tetrahedral meshes. We also support balancing mixed and other monotopological meshes (e.g., all quadrilaterals or all hexahedra).

ParMA's implementation relies on the PUMI parallel unstructured mesh infrastructure [31], and inter-process communication algorithms detailed by Ibanez et al. [29]. We refer readers to these papers for details on the element migration procedure and neighborhood communications for information exchange.

In this work, ParMA, combined with graph and geometric partitioning methods provided by Zoltan [16], satisfies the requirements for dynamic load balancing described in Section 2 to over one million parts on meshes with over 12 billion tetrahedral elements. Partition quality requirements 1 and 2 are satisfied by partitioning the mesh with a graph or geometric partitioner and then running ParMA to reduce the imbalance of mesh entity dimensions critical to the application. For example, ParMA is applied to balance the entities used as degree of freedom holders in finite element method procedures. The incremental partition change requirement (3) is implicitly satisfied by the definition of ParMA's diffusion procedure and recursive coordinate bisection. Graph-based methods provided by Zoltan's API also have execution modes that minimize data movement. Requirement 4 is implicitly satisfied as applications in the workflow are driven from the partitioning of the mesh that ParMA produces. Performance requirements 5 and 6 are satisfied by combining ParMA with a partitioner that scales to the required concurrency level. Lastly, requirement 7 is satisfied through Zoltan's API to interact with the mesh data structure and ParMA's direct use of mesh modification and query APIs.

3.1. Partitioned Mesh Representation. PUMI provides the $O(1)$ queries of intra- and inter- part mesh topology information needed by ParMA via a complete and distributed mesh representation [31, 54]. The distributed mesh is the union of mesh parts. A mesh part is defined as a collection of mesh faces M^2 in 2D, and regions M^3 in 3D, assigned to a processing resource, typically a core or hardware thread. Mesh entities are denoted as M_i^d, where d specifies the dimension and i specifies the id or index. At the shared boundary of two or more parts mesh entities are copied (as shown for mesh vertex M_0^0 and edge M_0^1 in Fig. 1) and locally tracked on each part through a remote copy object. Distributed mesh operations involving a mesh entity on the part boundary are coordinated through an ownership protocol; depicted by the discs and bold segments in Fig. 1.

Two parts with common boundary mesh entities are neighbors. Sets of mesh entities sharing common neighboring parts form a partition model entity [53]. Like mesh entities, we denote the ith partition model entity of dimension d as P_i^d. A mesh entity is classified on the partition model entity of equal or greater dimension which bounds it. For example, in Fig. 1 mesh vertex M_0^0 is classified on the partition model vertex P_0^0, mesh edge M_0^1 is classified on the partition model edge P_1^1, and mesh face M_0^2 is classified on the partition model face P_2^2. These classifications are respectively

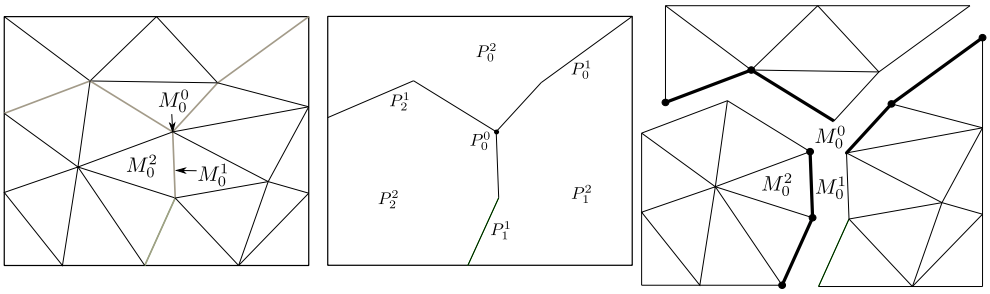


Figure 1: (left) Example of a mesh, (middle) its partition model, and (right) its ownership. Discs and bold segments denote entity ownership.

noted as $M_0^0 \sqsubset P_0^0$, $M_1^1 \sqsubset P_1^1$, and $M_0^2 \sqsubset P_2^2$. Information is exchanged by neighboring parts, typically for synchronizing data associated with part boundary entities, through non-blocking, collective, neighborhood communications provided by PCU [29, 42]. Using these communications, PUMI also provides procedures to efficiently move mesh elements between processors; referred to as migration.

3.2. Partition Improvement. ParMA reduces the peak imbalance of multiple entity dimensions by iteratively migrating some mesh elements from heavily loaded parts to neighboring parts with less load. The entity dimensions to balance are defined by an application specified priority list. For example, if element>vertex is specified then the algorithm prioritizes improvements to element balance over vertex balance. The greater-than relation indicates that element balance improvements are allowed to degrade the vertex balance, but vertex balance improvements cannot degrade the element balance. The balance of unlisted entity dimensions (edges and faces in this example) are not considered and may be degraded. If vertex=element is specified, then the algorithm considers the balance of mesh elements and vertices equally important. In this case, the lower-dimension entities are processed first as improvements to their balance tends to improve the balance of the entities they bound (higher-dimension entities). The target imbalance for each listed entity dimension is specified by the application as $tgtImb^d$ where $d \leq d_{max}$ (the maximum dimension entity in a mesh). Applications which perform work on entities regardless of their ownership define the imbalance of a part, I_p^d, as the weight of mesh entities of dimension d existing on part p divided by the average weight of dimension d entities per part. The weight of a mesh entity is set to one when it is not specified by the application. The maximum imbalance of dimension d entities across all parts is noted as I^d.

The ParMA iterative diffusion procedure is summarized in Algorithm 1. This process is repeated for each specified entity dimension in order of descending priority, as described above. For simplicity, the pseudo code is written with only a single entity dimension, d, being passed to the supporting procedures. In practice though, we have the list of higher priority entity dimensions to avoid disturbing the imbalance of the higher priority entities during the balancing of the current, lower priority, entity dimension. Iterations are stopped on line 9 if the target imbalance ($tgtImb^d$) is reached, or they are stopped on line 10 if no migration opportunities remain (discussed in Section 3.5), or if a maximum number of iterations is reached. Each diffusive iteration has four steps [61]. First, on line 2, neighboring parts exchange local information (e.g., the weight of mesh entities) using PCU. Next, each part determines how much

load needs to be migrated and where it needs to go on line 3, targetting, and then marks elements for migration on line 4, selection. Before migration is executed, on line 5, each part determines if too much weight is being sent to it, and, as necessary, cancels a portion of the incoming element migrations. The cancellation process is detailed in subsection 3.4.2. The final step, migration, moves the marked elements to their defined destinations using PUMI.

Algorithm 1 ParMA Load Balancing

```
1: procedure RunStep( (in/out) mesh, (in) d)
2:     ComputeAndExchangeWeights( (in) d, (out) weight, (out) neighborWeights)
3:     Targeting( (in) mesh, (in) weight, (in) neighborWeights, (out) targetWeights)
4:     Selection( (in) mesh, (in) d, (in) targetWeights, (out) migrationPlan)
5:     Cancellation( (in) mesh, (in) neighbors, (in/out) migrationPlan)
6:     Migration( (in/out) mesh, (in) migrationPlan)
7: procedure Balance( (in/out) mesh, (in) dimensions)
8:     for all d ∈ dimensions do
9:         while imbalance of d > tolerance do RunStep( (in/out) mesh, (in) d)
10:            if Balancing Stagnates then
11:                break
```

The targeting and entity selection steps are detailed in the following sections.

3.3. Targeting. ParMA defines the load transfer requirements for balancing a given entity dimension based on the relative weight of the entities in neighboring parts. Parts with an entity imbalance, I_p^d, greater than the specified imbalance, $tgtImb^d$, are defined as heavily loaded parts. A lightly loaded part is defined based on the partition improvement requirements. If the application requires vertex=edge>element then migration to decrease element imbalance should not increase the imbalance of vertices or edges. Thus, during element improvement a part is a 'lightly loaded' target to receive elements if it has fewer vertices, edges and elements than the heavy part.

The amount of load, l_{pq}^d, migrated from a heavily loaded part p to a neighboring part q during improvement of mesh entities of dimension d is defined as

$$l_{pq}^d = \alpha * sf * \left(\sum_i w(M_i^d \in p) - \sum_i w(M_i^d \in q) \right), \tag{1}$$

where $w(M_i^d)$ is the application specified weight associated with a given entity i, α is a diffusion rate limiting constant $\in (0, 1]$ [14], and, in 3D, sf is the ratio of mesh faces shared by parts p and q to the total number of faces classified on partition boundaries of p. The surface area bias sf helps define load transfer requirements that can be satisfied in a single iteration by selecting elements for migration that are classified on the part boundary. A large transfer across a small boundary will not only take several iterations to satisfy, it will also lead to a large increase in the number of entities classified on the part boundary as each iteration will 'tunnel' into the part. The entity selection process is detailed in Section 3.4.

We tested the effect of α on run time and imbalance to guide the choice of a conservative default value. The test mesh of the automotive part shown in Fig. 2 has 2048 parts and an initial vertex imbalance of 46%. Table 1 and Fig. 3 respectively show the run time and vertex imbalance as α is varied from 0.2 to 1.0. The target vertex imbalance was set to 5%. With the exception of the $\alpha = 1.0$ case, all the cases

reached an imbalance of 6% or 7% before stagnation detection stopped the vertex balancer (Section 3.5). Setting α to 0.6 yields the fewest iterations and the shortest run time. Increasing α from this value causes too many elements to be migrated in each iteration, which results in imbalance oscillations that increase the run time. Similarly, lower values of α increase the run time by migrating too few elements in each iteration. Given these observations, α is conservatively set to 0.5 for the remaining tests in this work. Note, this setting of α may be tuned for a specific case to improve performance.

Figure 2: Coarse mesh of the 2014 RPI Formula Hybrid suspension upright.

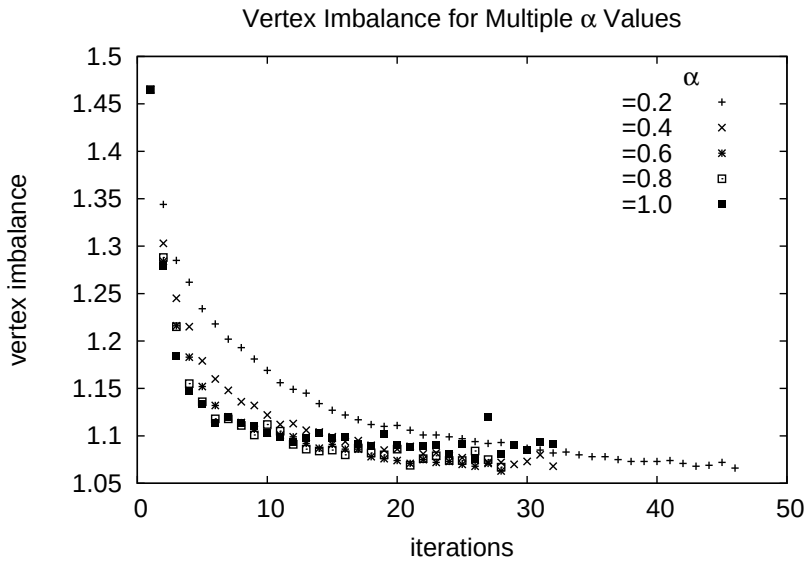


Figure 3: Effects of α on the number of iterations, and vertex imbalance. The initial mesh has 2048 parts and a 46% vertex imbalance.

Table 1: Diffusion iterations and run time for various α settings.

α	iterations	time (s)	I^0
0.2	45	19.3	1.07
0.4	31	14.0	1.07
0.6	27	12.5	1.06
0.8	27	13.5	1.07
1.0	31	16.6	1.09

Compared to Zhou's LIIPBMod, ParMA's use of Equation 1 enables finer grained migrations. In LIIPBMod, a part is a target for migration if (1) the difference between the vertex imbalance of the source part and the target part is greater than 2% or (2) the vertex imbalance is less than 4.5%. Note, LIIPBMod does not support weights associated with mesh vertices.

3.4. Entity Selection. Entity selection's primary objective is to reduce the imbalance of a given entity dimension. While selecting mesh elements for migration it is important to maintain inter-part boundaries with low surface area as an increase in the number of mesh entities classified on boundaries increases application communications, and in some cases, also the computational load [33]. Thus, entity selection's secondary objective is to reduce the number of mesh entities classified on partition model entities of dimension $d < d_{max}$.

Entity selection satisfies the objectives with part-level and entity-level heuristics. In Section 3.4.1 we describe how the part-level heuristic defines a vertex traversal order for evaluating the entity-level heuristic. Next, in Section 3.4.2, we describe how the entity-level heuristic evaluates the topology of a cavity; the set of elements adjacent to a given vertex. Combined, these two procedures reduce both the surface-to-volume ratio of the parts and their entity imbalance. Pseudo code for the selection procedure, as called in Algorithm 1, is given in Algorithm 2 and described in the following sections.

Algorithm 2 ParMA Selection

```
1: procedure SELECTION( (in) mesh, (in) d)
2:     if dist not set then
3:         IDENTIFYDISCONNECTEDCOMPONENTS((in) mesh, (out) comps)
4:         SETVERTEXCOMPONENTIDS((in) mesh, (in) comps, (out) ids)
5:         FINDTOPOLOGICALCENTERS((in) mesh, (in) comps, (out) centers)
6:         COMPUTECOREDISTANCE((in) mesh, (in) ids, (in) centers, (out) dist)
7:         OFFSETCOREDISTANCE((in) mesh, (out) dist)
8:     else
9:         UPDATEDISTANCE((in) mesh, (in/out) dist)
10:    for cavSize ∈ {2, 4, 6, 8, 10, 12} do
11:        CREATETRAVERSALQUEUE((in) mesh, (in) dist, (out) q)
12:        for all v ∈ q do
13:            if SHOULDMIGRATECAVITY((in) mesh, (in) v, (in) cavSize) then
14:                Add cavity of v to migrationPlan
```

3.4.1. Part-level Core Distance Heuristic. The number of mesh entities classified on partition model entities is reduced by migrating elements that are furthest from the topological center of the part, referred to as "the core". To find these

elements we traverse the part boundary vertices in order of their distance from the core. We define this distance as the shortest edge-based path between a vertex and the part's core. Thus, as diffusive iterations are executed, elements bounding vertices far from the core are migrated and the maximum distance of the part is reduced [18, 39]. This approach satisfies the second entity selection objective by forming parts with lower surface to volume ratios and reduced communications. In Algorithm 2 the core is found on line 5 and the distance is computed on line 6.

To understand the distance computation procedures, we must first account for parts produced by the graph and geometric partitioning that have multiple connected components. We define a (connected) component of a part as the set of elements in which there exists a path via M^{d-1} adjacencies (faces in 3D) between any two elements. Given this complexity, we first identify the components (lines 3 and 4 of Algorithm 2), compute the distance in each component (lines 5 and 6 of Algorithm 2) , and then offset the component distances to ensure a strictly increasing ordering for the traversal of boundary vertices (line 7 of Algorithm 2); we want the traversal to process the entire boundary of one component before moving on to the next one. The remainder of this subsection defines these procedures.

Connected components are identified via a breadth-first M^{d-1} adjacency-based traversal [13] starting at the first mesh element in the part (based on iterator ordering). As elements are visited, they are marked with the component id. When there are no more unmarked M^{d-1} adjacent elements to visit, the component id is incremented and the traversal is restarted with an unmarked element in another component. This process is repeated until all elements in the part are marked with a component id.

By traversing M^{d-1} mesh adjacencies between elements we have identified components with the strongest topological connectivity. But, to compute the core distance at mesh vertices, we first need to uniquely assign vertices to components. For vertices bounded by elements with the same component id the assignment is obvious. The problem comes with vertices at the common boundaries between components formed by lower dimension topological mesh adjacencies (i.e., an edge or vertex adjacency). To resolve this assignment issue, we set the vertex id to the lowest bounding component id. Now that vertices have component ids, we can find the vertices at the topological center of each component.

We find the central vertices in a component via a breadth-first traversal starting from all the boundary vertices of a component. When there are no more vertices to visit the traversal ends. From the set of vertices with the largest traversal depth, the first (based on vertex iterator ordering) is chosen as the component's core. The left half of Fig. 4 shows the vertices marked with their traversal depth. Note that selecting a different vertex with a depth of three could reduce the maximum distance to any boundary vertex, thus representing a more central vertex, and result in a small improvement to the subsequent boundary traversal. From the central vertices Dijkstra's algorithm [12] is run to compute the core distance to all other vertices in the component. The core distance at each vertex is shown in the right half of Fig. 4. A more complex example of distancing is shown in Fig. 5.

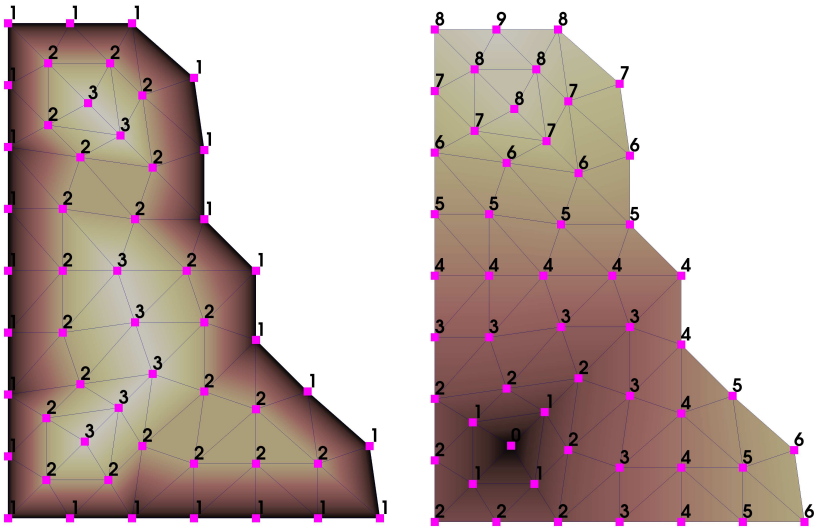
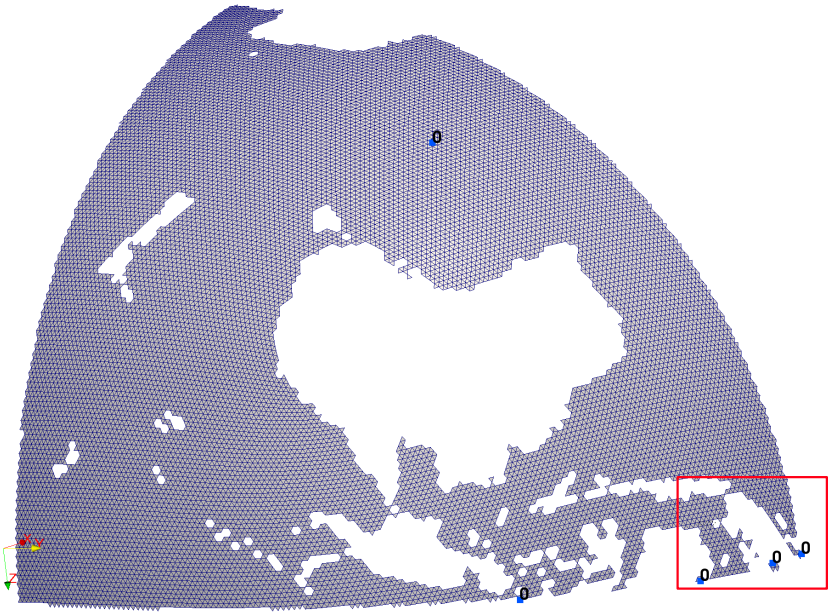
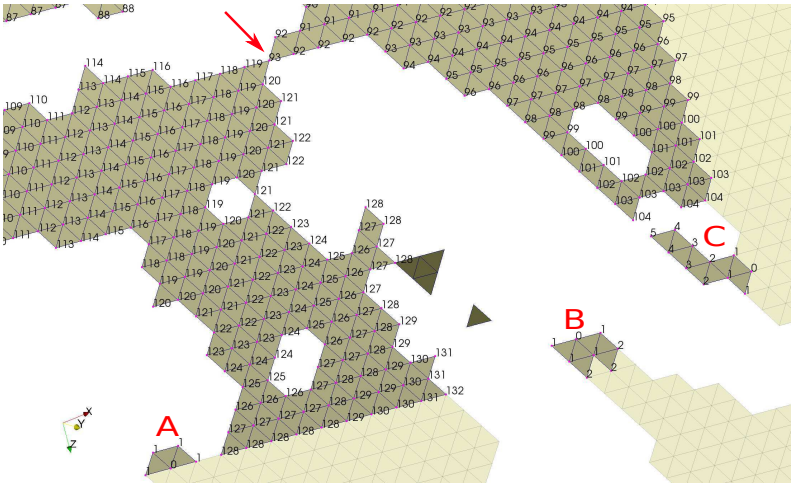


Figure 4: (left) The distance from each vertex to the boundary and (right) the distance from the core vertex (marked with a zero near the bottom left corner).



(a) Component core vertices marked with a zero.



(b) An edge-disconnected junction (arrow) and three disconnected components (A, B, C).

Figure 5: Components in one of four parts of the MPAS 60km [32] ocean mesh. Dark shaded elements are isolated (no M^{d-1} adjacency path to elements on the part boundary) and light shaded elements are on a different part.

Now that all components have vertices with distance, we must offset the distances so that our element selection procedure can traverse all the boundary vertices of a component before moving to another component. In Algorithm 2 the offset is computed on line 7. Fig. 6 depicts the distance of the disconnected components before

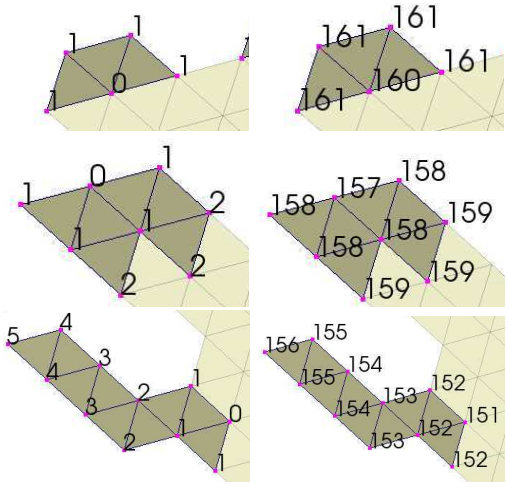


Figure 6: Core distance of disconnected components A, B, and C from Fig. 5 (left) before, and (right) after, the offset is applied.

and after the offset is applied. Algorithm 3 computes the component vertex distance offsets. The procedure begins by sorting the components in order of descending depth; forming the list c. Next, on line 2, the deepest component, r_0, has its offset set to zero. Lines 3 through 4 then compute the offset of the i^{th} component, r_i, by summing the previous component's offset and maximum distance, $r_{i-1} + max(R(M_j^0 \in c(i-1)))$, (where $R(M_j^0)$ is the distance of a vertex), plus an upper bound on a component's distance increase, $maxDistIncrease$. This upper bound enables fast distance updates by including a buffer into the offset that allows the parts to grow during diffusion iterations without overlapping. As each diffusion iteration can only add one layer of elements to a component, the maximum growth in distance for a component is bounded by the number of iterations. So, $maxDistIncrease$ is set to the maximum number of diffusive iterations. The final step on line 5 loops over the components in ascending order of their depth and applies the offset to their vertices. This component traversal order, combined with the conditional checking that the current distance value is less than the offset, prevents the distance of vertices on the boundary of two components being offset multiple times.

Algorithm 3 Vertex Distance Offset

1: c ← sortDescending(components)
2: $r_0 \leftarrow 0$ //component zero's offset
3: **for** $i \leftarrow 1$, $numComponents$ **do**
4: $\quad r_i \leftarrow r_{i-1} + max(R(M_j^0 \in c(i-1))) + 1 + maxDistIncrease$
5: **for** $i \leftarrow numComponents$, 1 **do**
6: $\quad$ **for** $M_j^0 \in c(i)$ **do**
7: $\quad\quad$ **if** $R(M_j^0) < r_i$ **then**
8: $\quad\quad\quad R(M_j^0) \leftarrow R(M_j^0) + r_i$;

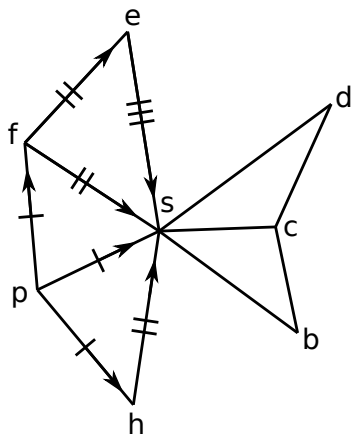


Figure 7: Determining if s is a non-manifold component boundary by creating a cavity of elements bounded by s, and then trying to walk from its parent vertex p to b, c, or d without going through s. The hash marks indicate the depth of each edge visited in the walk.

Within a component, detection of non-manifold [65] portions of the boundary is critical to ensure that the core distance accurately records the shortest M^{d-1} adjacent path from the core to each vertex. For example, consider the 2D non-manifold vertex junction indicated by the arrow in Fig. 5b. Here the paths from the core vertex marked in the upper portion of Fig. 5a to either side of the junction will have significantly different lengths due to the large holes in the mesh formed by land masses. Detection of a non-manifold junction at a given boundary vertex, s, is through the breadth-first traversal of s's cavity vertices (i.e., the vertices bounding elements in the cavity), rooted at the distance-1 parent of s. Vertices in the cavity are reachable via M^{d-1} adjacencies if the traversal can visit them without passing through s. For example, consider vertex s in the cavity depicted in Fig. 7 to have the lowest distance in the priority queue of vertices being processed by Dijkstra's algorithm. The detection traversal starts at vertex p, the parent of s, by enqueuing vertices f and h. s is also edge-adjacent to p, by definition, but it is skipped as paths through it are not considered. The traversal continues by dequeuing a vertex and enqueuing its edge-adjacent vertices that have not been previously visited and are not s. Fig. 7 depicts the depth of each edge in the traversal tree with hash marks. If there existed another element that was adjacent to s that was also adjacent to e and d, or h and b, then the edge (e, d) or (h, b) would provide an edge-adjacent path from p to b, c and d and the junction would be identified as manifold.

Compared to LIIPBMod, our part-level heuristic supports improvement of lower quality partitions by directly accounting for connected components, and non-manifold junctions within components.

In LIIPBMod, the boundary vertices are iterated over based on the order they appear in the underlying data structure without consideration for the part topology.

3.4.2. Entity-level Cavity Heuristics. In the previous subsection we described how the part-level heuristic defines a vertex traversal order for evaluating entity-level heuristics. In this subsection, we define those entity-level heuristics and

how they select elements for migration to reduce the entity imbalance. We start by describing size-based cavity selection. Next, we describe and demonstrate how multiple boundary traversals with increasing cavity size limits benefit partition improvement. In Algorithm 2 these steps are listed on lines 10 through 13. Lastly, we detail cancellation; a critical mechanism for multi-criteria load balancing.

Our entity-level, gain-like heuristic [19, 36] is based on Zhou's cavity-based approach [70, 72], but is more flexible. Like LIIPBMod, we check the number of elements in the cavity (the set of elements adjacent to a vertex on a part boundary), but we also check the adjacencies within the cavity, and the on- and off-part adjacencies external to the cavity. With this additional information we can migrate cavities that are bounded by vertices classified on partition model vertices, edges, and faces. LIIPBMod's heuristic avoided multi-part junctions; any cavity whose bounding vertex is classified on a partition model vertex or edge was not eligible for migration. In addition to more flexible migration, our heuristics improve the selection quality with (1) multiple boundary traversals with increasing cavity size in a single iteration, and (2) support for migrations to be canceled by the receiver.

The primary check for selection is based on the number of elements in a cavity. If a cavity is small, then migrating it will decrease the number of entities in the source part and classified on partition model entities. Conversely, migrating a face-connected cavity (i.e., between any two elements in the cavity there exists a path via face adjacencies) with several elements can result in an increase in the number of mesh entities classified on partition model entities. However, migrating small cavities with a few disconnected elements can yield significant entity reductions. Note, LIIPBMod uses a fixed cavity size of five elements.

To illustrate the effect of size and connectivity on entity reductions consider the cavities depicted in Fig. 8 and the reductions listed in Table 2. Fig. 8 (a-c) and (d-f) respectively depict face-connected and face-disconnected cavities. Here, the vertices bounding the cavities are marked with a disc. Vertices classified on the partition model face P_j^2 bounded by parts P_0^3 and P_1^3 are marked with a circle or disc, and in (c) a vertex classified on a partition model region, $M_i^0 \sqsubset P_0^3$, is marked with a square. In this example, all elements are migrated from P_0^3 to P_1^3. After migration the faces bounded by the circled vertices are now on the part boundary between P_0^3 to P_1^3, and in cavities (a-b) and (d-f) there is one less vertex classified on P_0^3. Migration of cavity (c) does not change the number of entities classified on the part boundary since there is an entity added to the part boundary for each one migrated.

Table 2: Reduction in the number of mesh entities classified on P_j^2 for cavities (a-f) depicted in Fig. 8.

	Cavity					
Entity Dimension	a	b	c	d	e	f
Vertex	1	1	0	1	1	1
Edge	3	2	0	5	7	8
Face	2	2	0	4	6	6

Ideally, we would like to select the combination of cavities for migration that results in the greatest imbalance reductions. Solving this problem exactly would be expensive, so instead, we iterate over the part boundary multiple times in order of descending vertex distance while relaxing (increasing) the cavity size selection limit before executing the PUMI element migration procedure. Thus, the first traversal of

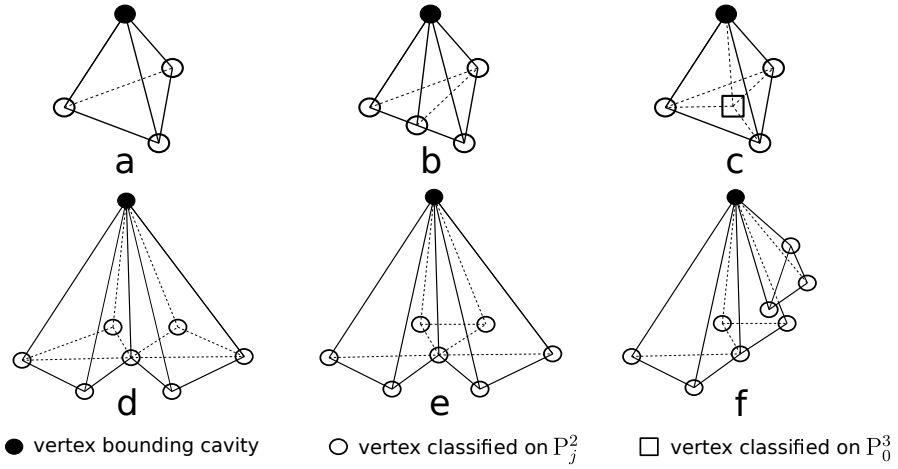


Figure 8: Vertex bounded cavities being migrated from part 0 to 1.

the boundary will select only cavities with one or two elements, followed by cavities with less than four elements in the second traversal (the first traversal may have created new one or two element candidates), and so on. The traversal stops at a cavity size limit of 12; roughly half of the average number of elements adjacent to a vertex in a tetrahedral mesh [2].

We tested the effectiveness of selection with an increasing size limit versus a static size limit by balancing a small test mesh. For both approaches the cavity size limit is set to 12. The test mesh of the suspension upright has 228 thousand elements and is partitioned to 2048 parts using RIB. The RIB partition has a perfect element balance and a 53% vertex imbalance. Our runs with vertex balancing ParMA targets a 5% vertex imbalance. Balancing with the increasing cavity size limit requires 2.0 seconds on 2048 Blue Gene/Q cores. At the end of the run, the target vertex imbalance is reached, the element imbalance is 9%, and the average number of vertices per part is reduced by 3.4%. On the same number of cores, the fixed cavity size run takes 3.4 seconds to reach the target vertex imbalance and has a 15% element imbalance, and a slight (0.07%) increase in the average number of vertices per part.

Once a cavity is selected, it needs to be assigned to a neighboring part for migration. The assignment and subsequent migration should result in a reduction of the number of mesh entities classified on the part boundary. In a 3D mesh we assign the cavity to the part that shares the most mesh edges with it. Counting shared edges avoids counting vertices (the lowest dimension shared entity) that are not adjacent to a higher dimension shared entity (an edge or a face) while providing more information than the counting of shared faces (the highest dimension shared entity, in 3D). Fig. 9 depicts a two element cavity with entities classified on both partition model faces and edges. Specifically, the cavity has two faces shared with part one (dark shaded), two faces with part two (unshaded), and an additional classification of edge F on the partition model edge shared with part two (dashed line in bold). Counting shared edges correctly identifies part two as the destination; it has six cavity edges versus part one only having five. The 'Sum' row of Table 3 lists the total cavity edge count on each part when the cavity elements are owned by part zero, the initial owner, and parts one and two, the two possible target parts. For this example, migrating the

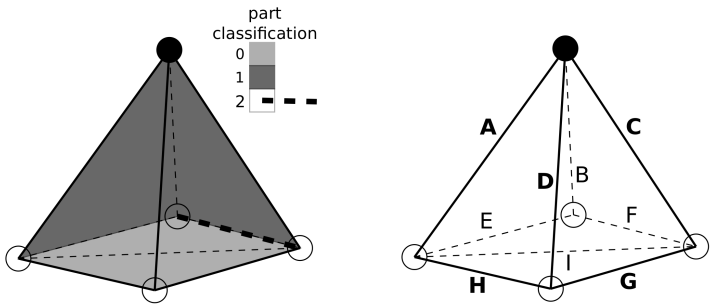


Figure 9: Counting mesh entity partition model classification to select either part one or part two for the migration of part zero cavity elements. The cavity bounding vertex is marked with a disc. (left) Part classification; faces in the foreground classified on part 2 are not shaded. (right) Mesh edge labels. For clarity, edges in the foreground have bold labels.

Table 3: Existence of Fig. 9 cavity edges on parts. The column groups list the edge existence prior to migration of the cavity (Owner=0), and after migration to part N (Owner=N). An entry is '1' if the edge exists on the part. The last row lists the total number of cavity edges on each part.

		Cavity Owner								
		0			1			2		
	Part	0	1	2	0	1	2	0	1	2
Edge	A	1	1	1	0	1	1	0	1	1
	B	1	1	0	0	1	0	0	1	1
	C	1	1	1	0	1	1	0	1	1
	D	1	0	1	0	1	1	0	0	1
	E	1	1	0	1	1	0	1	1	1
	F	1	1	1	1	1	1	1	1	1
	G	1	0	1	1	1	1	1	0	1
	H	1	0	1	1	1	1	1	0	1
	I	1	0	0	1	1	0	1	0	1
	Sum	9	5	6	6	9	6	5	5	9

cavity to part two reduces the total number of shared edges from 20 to 19; if part one were selected the total number of shared edges would increase by one. If multiple parts are tied for the most shared edges then the first part with remaining capacity is selected as the destination.

As the part boundary is traversed and the cavity heuristic selects elements for migration, the weight of the selected entities is tracked to prevent migrating too much weight to the target parts. Tracking is based on the simple rule, rooted in the unique assignment of elements to parts, that an entity will not exist on the part if all the elements it bounds are marked for migration. Thus, the weight tracking mechanism checks for this condition, and if satisfied, adds the entities weight to the running total for the given destination part.

During the balancing of lower priority entity dimensions (e.g., elements during vertex > element balancing) the imbalance of higher priority entity dimensions is

preserved by canceling the migration of some elements [49]. First, the sending parts determine how much weight associated with higher priority entities is migrated to the target parts. These weights are then sent to the respective targets using PCU's neighborhood communication procedures [29]. The target part then iterates over the incoming migration requests in descending order of the migration weight, accepts the request if capacity remains, reduces the remaining capacity accordingly, and sends the accepted weight to the sender. The sending part then traverses the list of migration elements in the order they were selected (i.e., descending distance from the parts topological core), and keeps elements in the list until the peer's higher priority entity weight capacity is exceeded. A summary of the interaction between the part-level and entity-level heuristics is given in Section 3.6.

3.5. Stagnation Avoidance. A stagnation [70] avoidance procedure stops execution of diffusion when the imbalance or part shape has not improved over several iterations. Specifically, a second order accurate backward finite difference [21] approximates the rate of change of the imbalance, *imb*, and the average number of boundary mesh vertices per part, *sides*. Diffusion is stopped if the rate of change in *imb* is less than one percent of the target imbalance and the change in *sides* is less than one-hundredth of the initial *sides*.

3.6. Time Complexity. The part-level heuristic requires first executing an $O(|M^d|+|M^{d-1}|)$ element-based, breadth-first traversal to identify disconnected components. Next, the vertex component ids are set, $O(|M^0|)$, and boundary vertices are inserted into STL sets, $O(|M^0|log|M^0|)$. The component vertices are then traversed in breadth-first order via edge adjacencies to locate the topological center, $O(|M^0|+|M^1|)$. For simplicity, our implementation uses an STL set to maintain the vertices at each tree-depth. This choice adds $O(|M^0|log|M^0|)$ to the cost and could be avoided with a list-based traversal. Next, Dijkstra's algorithm is run to compute vertex distances, $O(|M^1|+|M^0|log|M^0|)$. As the vertices are visited, adjacent elements are accessed for non-manifold topology detection; a cost increase of $O(|M^d|)$. Lastly, the vertex distances are offset, $O(|M^0|)$. The overall complexity of the part-level heuristic for a 3D tetrahedral mesh is $O(|M^1|+|M^2|+|M^3|+|M^0|log|M^0|)$. But, for each entity dimension being balanced these procedures only need to be executed once. In subsequent iterations we can execute a lower cost distance update on just the boundary vertices (line 9 of Algorithm 2).

In each iteration the entity-level heuristic first requires building a STL map-based distance queue of vertices to traverse, $O(|M^0|log|M^0|)$. The vertices in the queue are then traversed, $O(|M^0|)$, and cavities constructed by adjacent element queries, $O(|M^d|)$. Lastly, the cavity edges are queried for determining the destination part, $O(|M^1|)$. Thus, the entity-level heuristic's complexity is $O(|M^0|log|M^0|+|M^1|+|M^d|)$.

A detailed analysis of convergence and overall time complexity of general diffusive load balancing procedures can be found in the work of Subramanian [61] and Berenbrink [3].

4. Results. ParMA support for balancing 2D and 3D unstructured meshes with complex topological features are demonstrated in the following subsections. First, we compare ParMA against its predecessor LIIPBMod on up to 256Ki ($256{\cdot}2^{10}$) parts. We then test the effect of ParMA's entity selection features described in Section 3.4 on partition quality and imbalance. Then, we present ParMA's ability to improve partitions created with graph and geometric partitioning methods on up to 1Mi (2^{20})

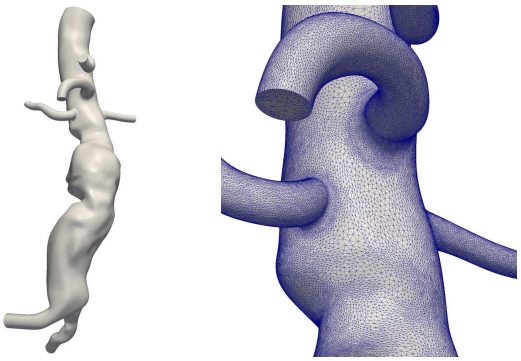


Figure 10: Abdominal aortic aneurysm (AAA) geometric model and close-up view of a coarse mesh.

parts. Lastly, we discuss the effect of partition improvement on the scalability of PHASTA computational fluid dynamics up to 512Ki parts.

4.1. LIIPBMod Comparison. We compare the performance of ParMA vertex>element improvement against LIIPBMod on a 64Ki, 128Ki, and 256Ki partition of a 941 million element tetrahedral abdominal aortic aneurysm (AAA) mesh. This mesh was generated by successively refining the initial coarse mesh shown in Fig. 10. Three test partitions of the mesh were created by running local ParMETIS (one instance per process [70]) part k-way on a 16Ki base partition created with global ParMETIS part k-way. Our partition improvement test then executed ParMA and LIIPBMod on the three partitions using the Mira Blue Gene/Q system at the Argonne Leadership Computing Facility (ALCF).

Fig. 11 depicts the change in vertex and element imbalance resulting from ParMA and LIIPBMod. In these tests LIIPBMod targets a 5% vertex imbalance and ParMA targets 5% vertex and element imbalance. Note that LIIPBMod does not explicitly target reducing the element imbalance; it simply tries not to harm it significantly while balancing vertices. LIIPBMod balancing stagnates at around 10% for the vertex imbalance and, at 256Ki, increases the element imbalance by two percentage points. At all three partition sizes ParMA meets the vertex and element imbalance target of 5% and executes 75% faster than LIIPBMod. For these partitions ParMA and LIIPBMod have an insignificant effect (less than one percent) on the total number of vertices. The ParMA features that support fast balancing are discussed in Sections 3, 3.3, and 3.4. Next, we discuss the performance cost and partition quality improvements of these features.

4.2. Feature Tests. We tested ParMA vertex>element improvement on a 497,058 triangular-element MPAS North America 15km-to-75km graded ocean mesh partitioned to 1Ki parts. The initial partition generated with local ParMETIS part k-way has a vertex and element imbalance of 36% and 17%, respectively, and on average, 280 vertices per part.

Configuration 1 of Table 4 serves as the baseline for feature inclusion. It uses iterator-based part boundary vertex traversal (disabled graph distance), disables detection of non-manifold part junctions, has a fixed cavity size for selection, and when balancing elements, does not cancel selections to help preserve vertex imbalance. Con-

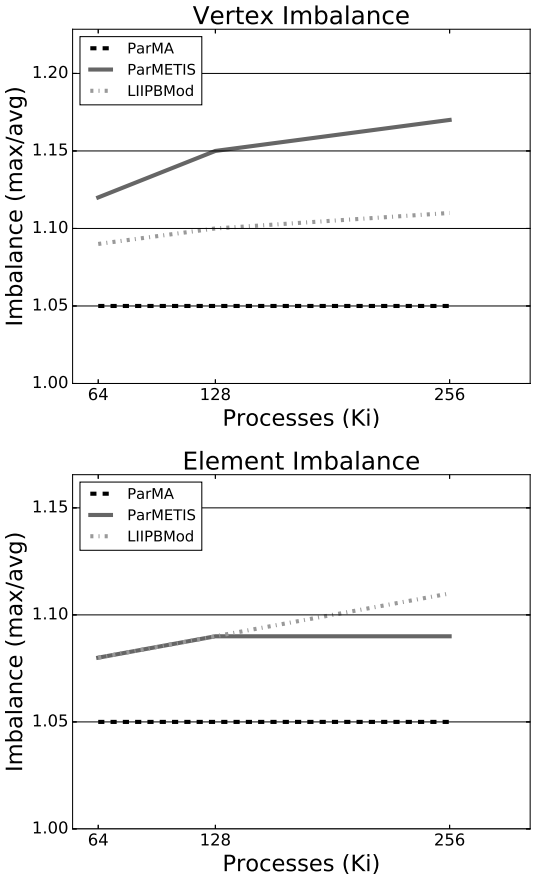


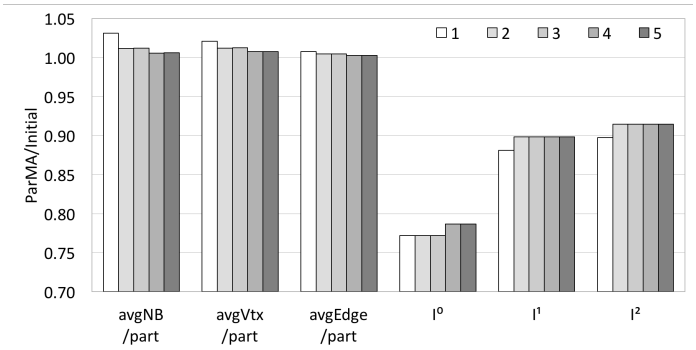
Figure 11: Evolution of the (top) vertex and (bottom) element imbalance with ParMA, LIIPBMod, and ParMETIS in the 941 million element AAA mesh.

figurations 2, 3, 4, and 5 successively add the features listed in Table 4.

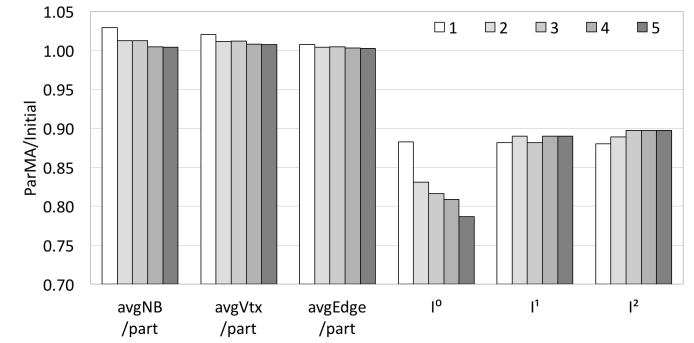
For each configuration Fig. 12a and Fig. 12b depict the change in partition quality, relative to the initial partition, after ParMA balancing. ParMA's target imbalance was set to 5% for vertices and elements. Partition quality is measured in three ways: (1) the average number of neighbors per part, counted via shared vertices, 'avgNB/part', (2) the average number of vertices and edges per part, 'avgVtx/part' and 'avgEdge/part', and (3) the entity imbalance, I^d. For each of these measures a value of one indicates no change from the initial partition, while a value greater (lower) than one indicates an increase (decrease) in the measure relative to the initial partition.

Fig. 12a depicts the improvement in quality after vertex balancing. The average number of neighbors, vertices, and edges per part increases by one percent or less with all features enabled. Relative to the over 20% decrease in vertex imbalance, these increases are negligible.

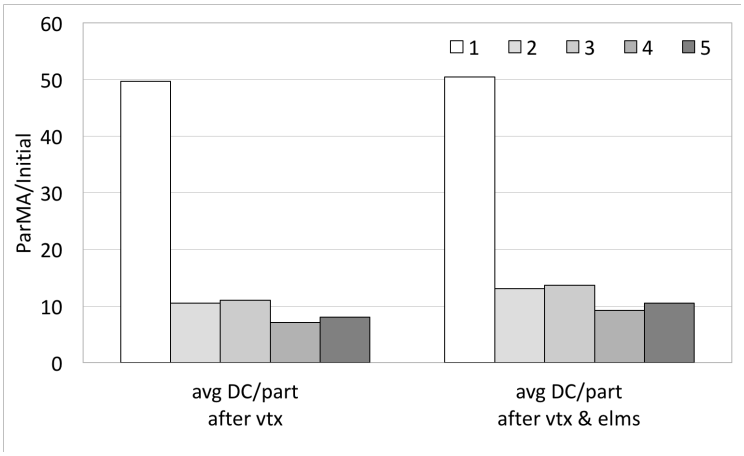
Vertex>element balancing, Fig. 12b, further improves the partition quality as features are enabled. After Configuration 1 (element balancing with no features enabled),



(a) Partition quality after vertex balancing.



(b) Partition quality after vertex > element balancing.



(c) Disconnected components.

Figure 12: Partition quality of a 1,024 part MPAS North America 15km to 75km graded ocean mesh using the ParMA configurations listed in Table 4.

Table 4: ParMA test configurations.

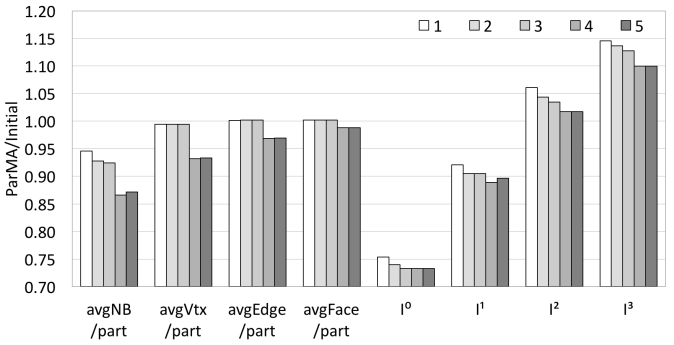
Configuration	Enabled Features
1	None
2	1 + core distance traversal
3	2 + non-manifold feature detection
4	3 + increasing cavity size selection
5	4 + selection cancellation

the element imbalance is reduced from 5% to 3% at the cost of a vertex imbalance increase from 5% to 20%. Enabling core distance traversal, Configuration 2, reduces the average number of disconnected components per part. Fig. 12c shows that the disconnected component count, relative to the initial partition, increases by 50x in Configuration 1 while Configuration 2 only has a 10x increase. The large reduction in disconnected components reduces the number of vertices on the part boundaries. This reduction in turn helps limit the vertex balance increase to 13% after element balancing. The features of Configuration 4 further reduce the number of boundary vertices (as indicated by the reductions in average neighbors, edges, and vertices per part) and results in a 10% vertex imbalance. With all features enabled, Configuration 5, a final vertex imbalance of 7% is reached while maintaining the 5% element imbalance and further improving the other quality measures. This Configuration runs in 72% of the time of Configuration 1; 3.07 seconds versus 4.25 seconds. The faster run time is mainly the result of vertex balancing times reducing from 4.09 seconds to 2.82 seconds, and only a slight increase in the element balancing times from 0.16 seconds to 0.25 seconds.

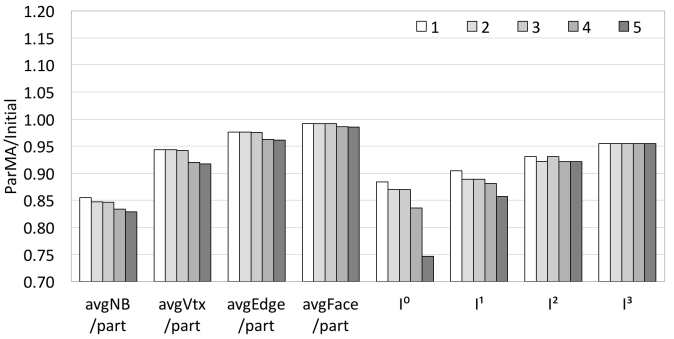
We also ran the feature test on the 3D 2.3 million element RPI Formula Hybrid suspension upright mesh, the geometric model depicted in Fig. 2. The test mesh has 2,048 parts and a 46% vertex and 10% element initial imbalance. ParMA's target imbalance was set to 5% for both vertex and vertex>element balancing. Fig. 13 shows the results of the tests. In Configuration 1 balancing the mesh vertices to 10% increases the element imbalance to 26%. The subsequent element balancing reduces the element imbalance to 5% in 17.8 seconds, but increases the vertex imbalance to 29%. As features are enabled the partition quality and imbalances improve at the cost of increased run time. Running with all features enabled (Configuration 5) requires 37.6 seconds (two times longer than Configuration 1), and reaches an element imbalance of 5% and a vertex imbalance of 9%.

A critical difference of the 3D upright tests to the 2D MPAS tests is the large reduction in disconnected parts and the related decrease in the average neighbors and entities per part. Compared to the initial partition, Configuration 5 of the upright test reduces the average neighbors, vertices, and disconnected components per part by 17%, 8%, and 93% respectively, and 3%, 2%, and 27% versus Configuration 1. This difference is mostly due to the change from 2D to 3D and the increased connectedness of the geometric model that enables more migration opportunities; the MPAS mesh has multiple geometric surfaces which only share one or two vertices with other surfaces.

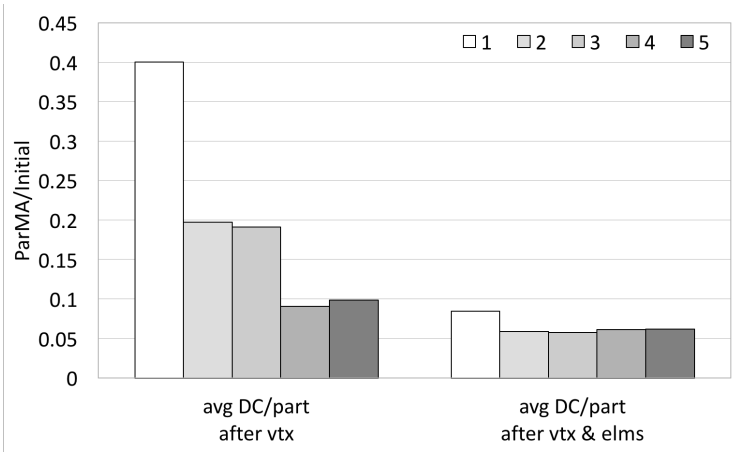
The feature tests were run using one part per core on the Blue Gene/Q at the Rensselaer Center for Computational Innovations. Tests with all features enabled and additional balancing criteria are described next.



(a) Partition quality after vertex balancing.



(b) Partition quality after vertex > element balancing.



(c) Disconnected components.

Figure 13: Partition quality of a 2,048 part RPI Formula Hybrid suspension upright mesh using the ParMA configurations listed in Table 4. Lower is better.

Table 5: vertex=edge>element partition improvement on a 2.3 million element, 2048 part, mesh of the RPI Formula Hybrid suspension upright of Fig. 2.

stage	avg/part vtx	avg/part edge	avg/part face	I^0	I^1	I^2	I^3	time (s)
adapt	**357.749**	**1741.012**	**2497.981**	**1.46**	**1.92**	**1.15**	**1.10**	
vertices	334.0	1687.7	2469.3	1.08	1.92	1.16	1.19	10.27
edges	330.5	1679.0	2464.3	1.09	1.06	1.09	1.13	6.88
elements	**328.829**	**1674.661**	**2461.637**	**1.09**	**1.06**	**1.07**	**1.08**	10.26
RIB	**350.457**	**1737.694**	**2503.369**	**1.53**	**1.92**	**1.10**	**1.00**	
vertices	337.8	1705.0	2483.4	1.04	1.95	1.07	1.10	3.01
edges	333.2	1692.8	2475.8	1.06	1.05	1.04	1.07	3.86
elements	**331.387**	**1687.526**	**2472.306**	**1.07**	**1.05**	**1.04**	**1.04**	0.85

4.3. Multi-Criteria Improvement. Analysis codes which have work associated with multiple entity dimensions and have a non-uniform distribution of that work require multi-criteria balancing. Codes with this requirement include finite elements with non-uniform p, particle-in-cell [69], contact/impact [20], atomistic-to-continuum [22], and other multi-model or multi-physics techniques [10]. ParMA satisfies this requirement by balancing the entity dimensions defined in a priority-sorted list. For each entity in the mesh the application also optionally provides weights specifying the associated computational load. To test this ability we ran ParMA vertex=edge>element balancing on a 2.3 million element, 2,048 part mesh of the suspension upright. The test emulates a non-uniform work distribution associated with edges by setting entity weights. On part zero edge weight is set to two; all other parts have entity weights of one.

Two initial partitions were used in testing; one is the result of mesh adaptation (listed as 'adapt'), and another is generated with RIB. The partitions' average entity counts and imbalances are listed in Table 5. The adapt partition, relative to the RIB partition, has a ten point higher element imbalance, and on average, four more neighbors and two more disconnected components per part. Given the lower initial quality, ParMA improvement on the adapt partition requires about 350% more time to run (27.4 seconds versus 7.7 seconds), and has final entity imbalances (noted in the 'elements' row) a few points higher than the final ParMA imbalances of the RIB partition. Note that, even with the run time increase, the time spent in ParMA is insignificant relative to the time spent executing a typical finite element analysis on a partition of this size.

Despite an initial weighted-edge imbalance of over 90% in both partitions, ParMA reduces the entity imbalances to less than 9% while also reducing the average per part entity weights by up to 5%. Critical to this result is ParMA's ability to diffuse away edge weight from the heavily imbalanced part zero while not overloading other parts. Diffusion reduces the number of mesh edges in part zero from 1674 to 901 in the adapt partition, and from 1665 to 889 in the RIB partition.

4.4. Partitioning to Over One Million Parts. ParMA quickly reduces large imbalances and improves part shape of a 1.6 billion element suspension upright mesh partitioned from 128Ki to 1Mi (2^{20}) parts (approximately 1500 elements/part). The initial 128Ki partition has less than 7% imbalance for all entity dimensions. We ran the tests on the Mira Blue Gene/Q located at the ALCF. One hardware thread was used per part.

- Partitioning with global RIB completes in 103 seconds and results in a 209% vertex imbalance and a perfect element imbalance. ParMA runs on 1Mi processors in 20 seconds and reduces the vertex imbalance to 6%, only increases the element imbalance to 4%, and reduces the average number of vertices per part by 5.5%.
- Local partitioning with ParMetis (one serial instance of ParMETIS for each initial part) completes in 9.0 seconds and results in a 63% vertex imbalance and a 12% element imbalance. ParMA runs in parallel on 1Mi processors in 9.4 seconds and reduces the vertex imbalance to 5%, the element imbalance to 4%, and reduces the average number of vertices per part by 2%.

Partitioning a 12.9 billion element mesh from 128Ki (< 7% imbalance) to 1Mi parts (approximately 12 thousand elements/part) using serial instances of ParMETIS completes in 60 seconds and results in a 35% vertex imbalance and an 11% element imbalance. Running ParMA in parallel on 1Mi processors takes 36 seconds to reduce the vertex and element imbalances to 5% and reduce the average number of vertices per part by 0.6%.

Table 6 lists the number of elements, the initial and target part counts, and the initial entity imbalances, I^{0-3} for vertices, edges, faces and regions, respectively, for three partitions. Table 7 lists the results of ParMA runs on those partitions. Note, the column 'dec. (%)' lists the percentage decrease in the average vertices per part after ParMA relative to the partitioning stage, 'Split'.

Table 6: Initial meshes for upright tests. The name of each is mesh is describing the number of elements in the target part.

name	elements	parts	target parts	elms per tgt. part	I^0	I^1	I^2	I^3
small	1.6×10^9	2^{17}	2^{20}	1541.7	1.06	1.06	1.06	1.07
medium	12.9×10^9	2^{17}	2^{20}	12 333.8	1.05	1.06	1.07	1.07
large	12.9×10^9	2^{17}	2^{19}	24 667.6	1.05	1.06	1.07	1.07

Table 7: X+ParMA vertex > element upright test results.

scope	density	method	stage	avg vtx	dec. (%)	I^0	I^1	I^2	I^3	tot (s)
local	small	rib	Split	455.1		1.34	1.18	1.13	1.13	10.67
			ParMA	427.6	6.42	1.07	1.06	1.05	1.05	8.94
		pmetis	Split	427.0		1.63	1.32	1.13	1.12	8.99
			ParMA	418.8	1.97	1.05	1.05	1.04	1.04	9.48
	medium	rib	Split	2825.5		1.31	1.14	1.08	1.07	54.32
			ParMA	2752.0	2.67	1.06	1.05	1.04	1.05	48.81
		pmetis	Split	2687.7		1.35	1.14	1.11	1.11	59.81
			ParMA	2671.3	0.61	1.05	1.05	1.04	1.05	36.15
	large	rib	Split	5273.9		1.16	1.13	1.12	1.13	42.69
			ParMA	5122.9	2.95	1.05	1.04	1.04	1.05	52.87
		pmetis	Split	5132.4		1.21	1.09	1.10	1.10	37.02
			ParMA	5102.2	0.59	1.04	1.04	1.04	1.04	41.55
global	small	rib	Split	470.1		**3.09**	2.07	1.45	1.00	103.14
			ParMA	445.4	5.54	**1.06**	1.04	1.03	1.04	20.23
	large	rib	Split	5367.3		2.49	1.70	1.29	1.00	96.79
			ParMA	5228.8	2.65	1.05	1.02	1.03	1.04	379.84

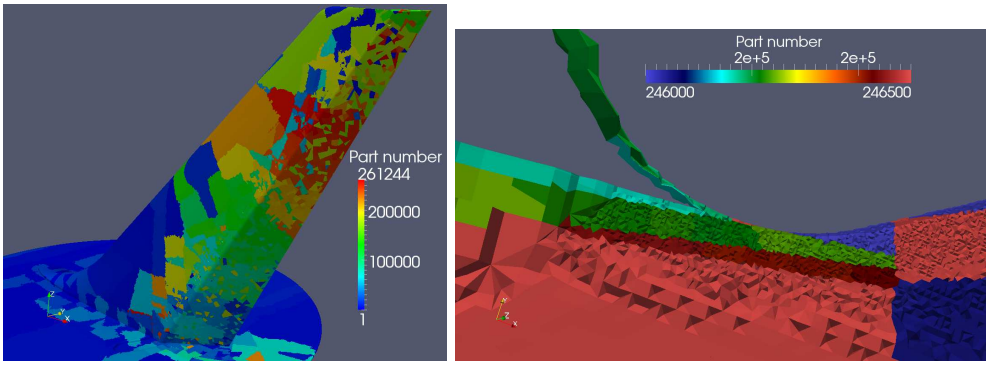


Figure 14: (left) Full view of the vertical stabilizer and rudder and (right) a slice at their junction colored by part number illustrating the complex geometry and small features of the fluid mesh.

4.5. CFD Scaling Improvement. As an example of ParMA's ability to improve simulations of very complex geometric models at extreme scale, consider the geometry shown in Fig. 14. The left side of the figure depicts the surface of the vertical tail and rudder while the right side provides a detailed view of a complex geometric junction. At this junction we show a close-up view of a clip-plane cutting through the very small gap between the vertical stabilizer and the rudder where many parts are contained. In this region several of the parts are "cutoff" from the surrounding geometry and have a limited number of neighbors to diffuse through for partition improvement.

The partitions of the 1.2 billion element tetrahedral mesh for this study were obtained through a series of steps. First, mesh adaptation was executed on a 4Ki part mesh using an error-based size field [11, 45]. To balance and partition this mesh, global ParMETIS part k-way [34] was executed to create an 8Ki part mesh. Starting from this 8Ki part mesh, with a 7% vertex imbalance and 1% element imbalance, ParMETIS part k-way was applied locally to each part to create partitions of the mesh in powers of two from 64Ki parts to 512Ki parts. These partitions were then balanced using ParMA vertex>element to create a second set of partitions.

The flow in this case is solved by PHASTA. PHASTA is a stabilized finite element analysis code [66] using an implicit solver. The code is written in FORTRAN and is parallelized with MPI. PHASTA's computational work is dominated by equation formation and equation solution. Both types of work are executed on the same partition of mesh elements [46]. An ideal partition will have balanced elements for equation formation work, and balanced vertices, the degree-of-freedom holder, for equation solution work. Furthermore, the partition will have parts with a low surface-to-volume ratio to limit the cost of neighborhood communications that exchange information on boundary vertices [44].

As shown in Fig. 15, through three part-count doublings, ParMA is able to improve the vertex imbalance with only insignificant increases in element imbalance. For example, in the largest partition, 512Ki parts, ParMA reduces the vertex imbalance from 54% to 6%, and only increases the element imbalance from 1.8% to 3%. As expected, the 1.2 percentage point increase in element imbalance has no effect on the nearly perfect scaling of equation formation (scaling factor, defined

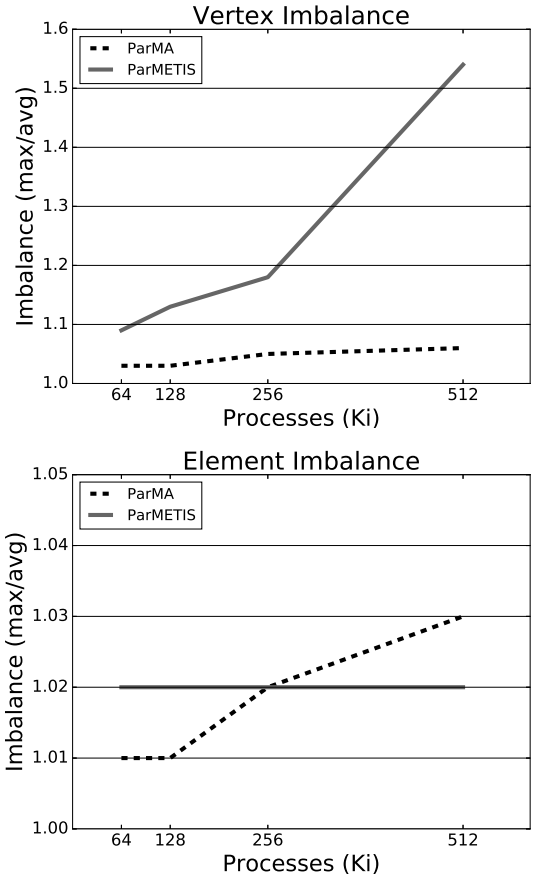


Figure 15: Evolution of the (top) vertex and (bottom) element imbalance with and without ParMA.

as $(time(base) \cdot procs(base))/(time(test) \cdot procs(test))$, of 0.96 maintained). Critically though, ParMA improves the linear algebra work performance by 28% over the ParMETIS partition, and improves scaling from 0.82 to 1.14, as shown in Fig. 16. As sparse linear algebra is memory bandwidth limited [71], a super-linear scaling is observed as the working data size is reduced and cache utilization is increased. Similar, but less dramatic, performance and scaling gains are observed in the 256Ki part case. In the smaller 64Ki and 128Ki partitions the performance difference is negligible. All PHASTA runs were performed on Mira using one process per core. This configuration, although not optimal for achieving peak floating point performance on the Blue Gene/Q, was selected to avoid unfortunate process to core mappings that could assign two heavily loaded processes to the same core, and thus confound the interpretation of performance results.

5. Conclusion. The ability to evenly distribute the work associated with a specific combination of mesh entity dimensions (i.e., vertices, edges, faces, and regions) in parallel unstructured mesh-based applications is critical to scalability on massively parallel leadership class systems. ParMA, partitioning using mesh adjacencies, cou-

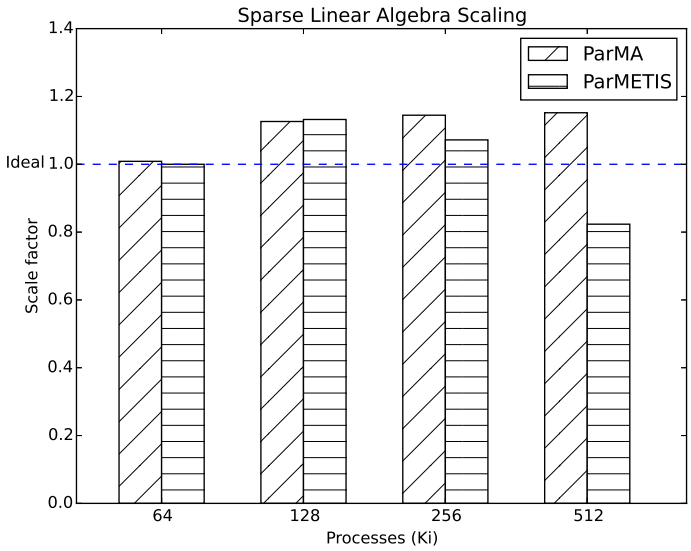


Figure 16: Improvement of PHASTA sparse linear algebra scaling with ParMA. The PHASTA performance on the 64Ki ParMETIS partition is used as a baseline for all runs. Higher is better.

pled with graph and geometric based partitioning tools, provides fast, multi-criteria, diffusive partition improvement to meet this need. Demonstrations are provided on meshes with over 12 billion elements running on over one million processes (four processes per core) on the Mira IBM Blue Gene/Q. Additionally, for a massively parallel PHASTA CFD analysis running on a half million processes (one process per core) on Mira, ParMA improves the performance of sparse linear algebra computations by 28% versus a ParMETIS partition. Likewise, the strong scaling factor of these computations is improved from 0.82 to 1.14; a critical result for efficiently reducing the time to solution as core count is increased. ParMA achieves these improvements by reducing the vertex imbalance from 54% to 6% while maintaing an element imbalance at or below 3%. Efforts to further understand the impact on additional partition quality metrics on the strong scalability of PHASTA, and other applications, using the latest leadership class systems are ongoing.

REFERENCES

[1] Cevdet Aykanat, B Barla Cambazoglu, and Bora Uçar, *Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices*, J. Parallel and Distributed Comput., 68 (2008), pp. 609–625.

[2] Mark W. Beall and Mark S. Shephard, *A general topology-based mesh data structure*, Int. J. Numerical Methods in Eng., 40 (1997), pp. 1573–1596.

[3] Petra Berenbrink, Tom Friedetzky, and Zengjian Hu, *A new analytical method for parallel, diffusion-type load balancing*, J. Parallel and Distributed Comput., 69 (2009), pp. 54–61.

[4] Marsha J Berger and Shahid H Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, Comput., IEEE Trans., 100 (1987), pp. 570–580.

[5] Rob H. Bisseling and Wouter Meesen, *Communication balancing in parallel sparse matrix-*

vector multiplication., ETNA. Electronic Transactions on Numerical Analysis, 21 (2005), pp. 47–65.

[6] Erik G. Boman, Karen D. Devine, Vitus J. Leung, Sivasankaran Rajamanickam, Lee Ann Riesen, Deveci Mehmet, and Çatalyürek Umit, *Zoltan2: Next-generation combinatorial toolkit*, Tech. Report SAND2012-9373C, Sandia Nat. Labs, Albuquerque, NM, USA, 2012.

[7] UV Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar, *UMPa: A multiobjective, multi-level partitioner for communication minimization*, Contemporary Math., 588 (2013), pp. 53–64.

[8] Umit V Çatalyürek and Cevdet Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, Parallel and Distributed Syst., IEEE Trans., 10 (1999), pp. 673–693.

[9] Umit V Çatalyürek, Erik G Boman, Karen D Devine, Doruk Bozdağ, Robert T Heaphy, and Lee Ann Riesen, *A repartitioning hypergraph model for dynamic load balancing*, J. Parallel and Distributed Comput., 69 (2009), pp. 711–724.

[10] C Chevalier, G Grospellier, F Ledoux, JC Weill, and France Arpajon, *Load balancing for mesh based multi-physics simulations in the Arcane framework*, in Proc. 8th Int. Conf. Eng. Computational Technol., Sept. 2012, pp. 47–62.

[11] Kedar C Chitale, Michel Rasquin, Onkar Sahni, Mark S Shephard, and Kenneth E Jansen, *Anisotropic boundary layer adaptivity of multi-element wings*, in 52nd Aerospace Sciences Meeting (SciTech). AIAA Paper, vol. 117, Jan. 2014, pp. 1–14.

[12] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction To Algorithms*, MIT Press, Cambridge, MA, USA, 2001.

[13] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, MIT Press, Cambridge, MA, USA, 2009, ch. 22.2.

[14] George Cybenko, *Dynamic load balancing for distributed memory multiprocessors*, J. Parallel and Distributed Comput., 7 (1989), pp. 279–301.

[15] M. Deveci, S. Rajamanickam, K. Devine, and U. Çatalyürek, *Multi-jagged: A scalable parallel spatial partitioning algorithm*, Parallel and Distributed Syst., IEEE Trans., 27 (2015), pp. 803–817.

[16] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications*, Comput. in Sci. Eng., 4 (2002), pp. 90–96.

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

[18] Ralf Diekmann, Robert Preis, Frank Schlimbach, and Chris Walshaw, *Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM*, Parallel Comput., 26 (2000), pp. 1555–1581.

[19] C.M. Fiduccia and R.M. Mattheyses, *A linear-time heuristic for improving network partitions*, in 19th Design Automation Conf., June 1982, pp. 175–181.

[20] J. Fingberg, A. Basermann, G. Lonsdale, J. Clinckemaillie, J.-M. Gratien, and R. Ducloux, *Dynamic Load Balancing for Parallel Structural Mechanics Simulations with DRAMA*, Civil-Comp Press, Edinburgh, UK, 2000, pp. 199–205.

[21] Bengt Fornberg, *Generation of finite difference formulas on arbitrarily spaced grids*, Math. of Computation, 51 (1988), pp. 699–706.

[22] Benjamin FrantzDale, Steven J. Plimpton, and Mark S. Shephard, *Software components for parallel multiscale simulation: an example with LAMMPS*, Eng. with Comput., 26 (2010), pp. 205–211.

[23] Norman E. Gibbs, William G. Poole, and Paul K. Stockmeyer, *An algorithm for reducing the bandwidth and profile of a sparse matrix*, SIAM J. Numerical Anal., 13 (1976), pp. 236–250.

[24] D. F. Harlacher, H. Klimach, S. Roller, C. Siebert, and F. Wolf, *Dynamic load balancing for unstructured meshes on space-filling curves*, in 26th Int. Parallel and Distributed Process. Symp. Workshops PhD Forum, May 2012, pp. 1661–1669.

[25] Bruce Hendrickson and Karen Devine, *Dynamic load balancing in computational mechanics*, Comput. Methods in Appl. Mech. and Eng., 184 (2000), pp. 485–500.

[26] YF Hu and RJ Blake, *An improved diffusion algorithm for dynamic load balancing*, Parallel Comput., 25 (1999), pp. 417–444.

[27] Y. F. Hu, R. J. Blake, and D. R. Emerson, *An optimal migration algorithm for dynamic load balancing*, Concurrency: Practice and Experience, 10 (1998), pp. 467–483.

[28] Thomas JR Hughes, *The finite Element Method: Linear Static and Dynamic Finite Element Analysis*, Courier Dover Publications, Mineola, NY, USA, 2012.

[29] Dan Ibanez, Ian Dunn, and Mark S. Shephard, *Hybrid MPI-thread parallelization of adaptive mesh operations*, Parallel Comput., 52 (2016), pp. 133–143.
[30] Daniel Alejandro Ibanez, *Conformal Mesh Adaptation On Heterogeneous Supercomputers*, PhD thesis, Rensselaer Polytechnic Inst., Troy, NY, 2016.
[31] Daniel A. Ibanez, E. Seegyoung Seol, Cameron W. Smith, and Mark S. Shephard, *PUMI: Parallel unstructured mesh infrastructure*, ACM Trans. Math. Softw., 42 (2016), pp. 17:1–17:28.
[32] Doug Jacobsen, Mark Petersen, Todd Ringler, and Michael Duda, *Mpas-ocean model user's guide. version 2.0*, 2014.
[33] Kenneth E. Jansen, Christian H. Whiting, and Gregory M. Hulbert, *A generalized-α method for integrating the filtered Navier-Stokes equations with a stabilized finite element method*, Comput. Methods in Appl. Mech. and Eng., 190 (2000), pp. 305–319.
[34] George Karypis and Vipin Kumar, *Multilevel algorithms for multi-constraint graph partitioning*, in Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC), IEEE, Nov. 1998, pp. 1–13.
[35] ———, *Parallel multilevel series k-way partitioning scheme for irregular graphs*, Siam Rev., 41 (1999), pp. 278–300.
[36] Brian W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, The Bell System Tech. J., 49 (1970), pp. 291–307.
[37] Dominique LaSalle and George Karypis, *Multi-threaded graph partitioning*, in Parallel & Distributed Process. (IPDPS), IEEE 27th Int. Symp., May 2013, pp. 225–236.
[38] Henning Meyerhenke, Burkhard Monien, and Stefan Schamberger, *Graph partitioning and disturbed diffusion*, Parallel Comput., 35 (2009), pp. 544–569.
[39] Henning Meyerhenke and Stefan Schamberger, *Balancing parallel adaptive FEM computations by solving systems of linear equations*, in Euro-Par Parallel Process., Aug.-Sept. 2005, pp. 209–219.
[40] R.M. O'Bara, M.W. Beall, and M.S. Shephard, *Attribute management system for engineering analysis*, Eng. with Comput., 18 (2002), pp. 339–351.
[41] Chao-Wei Ou and Sanjay Ranka, *Parallel incremental graph partitioning*, IEEE Trans. Parallel Distrib. Syst., 8 (1997), pp. 884–896.
[42] Aleksandr Ovcharenko, Daniel Ibanez, Fabien Delalondre, Onkar Sahni, Kenneth E. Jansen, Christopher D. Carothers, and Mark S. Shephard, *Neighborhood communication paradigm to increase scalability in large-scale dynamic scientific applications*, Parallel Comput., 38 (2012), pp. 140–156.
[43] Ekkehard Ramm, E Rank, R Rannacher, K Schweizerhof, E Stein, W Wendland, G Wittum, Peter Wriggers, and Walter Wunderlich, *Error-controlled Adaptive Finite Elements in Solid Mechanics*, John Wiley & Sons, West Sussex, England, 2003.
[44] Michel Rasquin, Cameron Smith, Kedar Chitale, E. Seegyoung Seol, Benjamin A. Matthews, Jeffrey L. Martin, Onkar Sahni, Raymond M. Loy, Mark S. Shephard, and Kenneth E. Jansen, *Scalable implicit flow solver for realistic wing simulations with flow control*, Comput. in Sci. & Eng., 16 (2014), pp. 13–21.
[45] Onkar Sahni, Jens Müller, Kenneth E. Jansen, Mark S. Shephard, and Charles A. Taylor, *Efficient anisotropic adaptive discretization of cardiovascular system*, Comput. Methods in Appl. Mech. and Eng., 195 (2006), pp. 5634–5655.
[46] Onkar Sahni, Min Zhou, Mark S Shephard, and Kenneth E Jansen, *Scalable implicit finite element solver for massively parallel processing with demonstration to 160k cores*, in Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC), IEEE, Nov. 2009, pp. 1–12.
[47] Erik Saule, Erdeniz Ö. Baş, and Ümit V. Çatalyürek, *Load-balancing spatially located computations using rectangular partitions*, J. Parallel and Distributed Comput., 72 (2012), pp. 1201–1214.
[48] Stefan Schamberger and Jens-Michael Wierum, *Partitioning finite element meshes using space-filling curves*, Future Generation Comput. Syst., 21 (2005), pp. 759–766.
[49] Kirk Schloegel, George Karypis, and Vipin Kumar, *Multilevel diffusion schemes for repartitioning of adaptive meshes*, J. Parallel and Distributed Comput., 47 (1997), pp. 109–124.
[50] ———, *Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes*, Parallel and Distributed Syst., IEEE Trans., 12 (2001), pp. 451–466.
[51] ———, *Parallel static and dynamic multi-constraint graph partitioning*, Concurrency and Computation: Practice and Experience, 14 (2002), pp. 219–240.
[52] ———, *Graph Partitioning for High-performance Scientific Simulations*, Morgan Kaufmann Inc., San Francisco, CA, USA, 2003, pp. 491–541.
[53] Eunyoung Seegyoung Seol, *FMDB: flexible distributed mesh database for parallel automated*

adaptive analysis, PhD thesis, Rensselaer Polytechnic Inst., Troy, NY, 2005.

[54] E. Seegyoung Seol, Cameron W. Smith, Daniel A. Ibanez, and Mark S. Shephard, *A parallel unstructured mesh infrastructure*, in Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC), IEEE, Nov. 2012, pp. 1124–1132.

[55] Mark S Shephard, M.W. Beall, R.M. O'Bara, and B.E. Webster, *Toward simulation-based design*, Finite Elements in Anal. and Design, 40 (2004), pp. 1575–1598.

[56] Mark S Shephard, Cameron Smith, and John E Kolb, *Bringing hpc to engineering innovation*, Comput. in Sci. & Eng., 15 (2013), pp. 16–25.

[57] H.D. Simon, *Partitioning of unstructured problems for parallel processing*, Comput. Syst. in Eng., 2 (1991), pp. 135–148.

[58] John Skilling, *Programming the hilbert curve*, in 23rd Int. Workshop on Bayesian Inference and Maximum Entropy Methods in Sci. and Eng., vol. 707, Aug. 2004, pp. 381–387.

[59] George M. Slota, Kamesh Madduri, and Sivasankaran Rajamanickam, *Complex network partitioning using label propagation*, SIAM Journal on Scientific Computing, 38 (2016), pp. S620–S645.

[60] G. M. Slota, S. Rajamanickam, Karen Devine, and K. Madduri, *Partitioning trillion-edge graphs in minutes*, in Int. Parallel & Distributed Process. Symp. (IPDPS), 2017.

[61] Raghu Subramanian and Isaac D Scherson, *An analysis of diffusive load-balancing*, in Proc. sixth Annu. ACM Symp. Parallel algorithms and architectures, June 1994, pp. 220–225.

[62] Valerie Taylor and Bahram Nour-omid, *A study of the factorization fill-in for a parallel implementation of the finite element method*, Int. J. Numer. Meth. Engng, 37 (1994), pp. 3809–3823.

[63] Saurabh Tendulkar, Mark Beall, Mark S Shephard, and KE Jansen, *Parallel mesh generation and adaptation for CAD geometries*, in Proc. NAFEMS World Congr., NAFEMS, May 2011, pp. 1–12.

[64] C Walshaw, Mark Cross, and MG Everett, *Dynamic mesh partitioning: A unified optimisation and load-balancing algorithm*, Tech. Report 95/IM/06, Univ. of Greenwich, London, UK, 1995.

[65] Kevin Weiler, *The radial edge structure: a topological representation for non-manifold geometric boundary modeling*, in Geometric Modeling for CAD Appl. First IFIP WG5.2 Work. Conf., May 1988, pp. 3–36.

[66] Christian H. Whiting and Kenneth E. Jansen, *A stabilized finite element method for the incompressible navier-stokes equations using a hierarchical basis*, Int. J. Numerical Methods in Fluids, 35 (2001), pp. 93–116.

[67] Marc H Willebeek-LeMair and Anthony P. Reeves, *Strategies for dynamic load balancing on highly parallel computers*, Parallel and Distributed Syst., IEEE Trans., 4 (1993), pp. 979–993.

[68] Roy D. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency: Pract. Exper., 3 (1991), pp. 457–481.

[69] Patrick H Worley, Eduardo D'Azevedo, Robert Hager, Seung-Hoe Ku, Eisung Yoon, and CS Chang, *Balancing particle and mesh computation in a particle-in-cell code*, in Proc. Cray Users Group Meeting, May 2016, pp. 1–10.

[70] M. Zhou, O. Sahni, K.D. Devine, M.S. Shephard, and K.E. Jansen, *Controlling unstructured mesh partitions for massively parallel simulations*, SIAM J. Scientific Comput., 32 (2010), pp. 3201–3227.

[71] Min Zhou, Onkar Sahni, Mark S Shephard, Christopher D Carothers, and Kenneth E Jansen, *Adjacency-based data reordering algorithm for acceleration of finite element computations*, Scientific Programming, 18 (2010), pp. 107–123.

[72] Min Zhou, Onkar Sahni, Ting Xie, Mark S Shephard, and Kenneth E Jansen, *Unstructured mesh partition improvement for implicit finite element at extreme scale*, The J. Supercomputing, 59 (2012), pp. 1218–1228.