Dynamic Load Balancing of Massively Parallel Unstructured Meshes

Gerrett Diamond, Cameron W. Smith, Mark S. Shephard
diamog@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York

ABSTRACT

Simulating systems with evolving relational structures on massively parallel computers require the computational work to be evenly distributed across the processing resources throughout the simulation. Adaptive, unstructured, mesh-based finite element and finite volume tools best exemplify this need. We present EnGPar and its diffusive partition improvement method that accounts for multiple application specified criteria. EnGPar's performance is compared against its predecessor, ParMA. Specifically, partition improvement results are provided on up to 512Ki processes of the Argonne Leadership Computing Facility's Mira BlueGene/Q system.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel and high-performance simulations**; • **Theory of computation** → *Massively parallel algorithms*; • **Computer systems organization** → Multiple instruction, single data; Single instruction, multiple data;

KEYWORDS

partition improvement,multigraph,hypergraph,dynamic load balancing

ACM Reference Format:
Gerrett Diamond, Cameron W. Smith, Mark S. Shephard. 2017. Dynamic Load Balancing of Massively Parallel Unstructured Meshes. In *Proceedings of 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, Denver, CO, USA, November 2017 (ScalA'17),* 7 pages.
https://doi.org/10.1145/3148226.3148236

1 INTRODUCTION

In evolving distributed simulations with complex relational structures the computational work needs to be evenly distributed throughout the simulation. In many applications, this requires starting with balanced partitions at the beginning of the simulation as well as the continuous rebalancing of the work as the simulation evolves. In these cases, re-running static partitioning algorithms is too expensive to be done repetitively. For better performance, efficient dynamic load balancing strategies must be applied to the evolving structure. A second challenge common to many simulations of interest arises from the fact that overall workload balance requires balancing multiple entity types, each with its own computational balance and communication criterion.

An efficient set of algorithms to solve these challenges use diffusive load balancing techniques. These algorithms perform load balancing across part boundaries and can be applied in succession to perform multiple levels of partitioning. One such diffusive load balancer, ParMA [23], works directly on an element-based unstructured mesh data structure called the Parallel Unstructured Mesh Infrastucture (PUMI) [16]. ParMA utilizes mesh adjacencies to perform localized balancing through diffusive migration from heavy parts to lighter neighbors. It has been shown that these operations can be efficiently used to perform multi-criteria load balancing in order to improve the partition that is returned by graph/hypergraph and geometric methods [25, 26].

While ParMA has been used to improve partitions for several simulations using mesh databases, there are applications that have different relation based data structures that have similar partitioning requirements. Two examples of these structures are scale-free graphs often used in computational social science to analyze social phenomena [13, 21] and vertex-based unstructured meshes which are sometimes used in computational fluid dynamic simulations [1]. While the latter example can be handled in ParMA by converting the vertex-based mesh to PUMI's element-based mesh, there are some challenges to appropriately account for the switch and it becomes more difficult to accurately approximate the load of each part. To extend ParMA's capabilities to these other applications, we present EnGPar. It utilizes an expanded graph structure to provide a general representation of relation-based data. Using this graph structure, a new implementation of ParMA's diffusive load balancing algorithm is given.

Section 2 defines notation. Section 3 gives an overview of partitioning techniques. In sections 4 and 5 EnGPar's design and implementation details are provided. Experiments and results are given in section 6 and 7. Finally, conclusions and future plans are discussed in section 8.

2 NOTATION

- M^d a set of mesh entities of dimension d.
- M_i^d the ith mesh entity of dimension d.
- V a set of vertices u_i which uniquely exist on one part such that $V = \bigcup_{\forall_i} u_i$.
- E_i a set of relations e_{ij} of type i that represent a edge between two vertices, $u, v \in V$ such that $E_i = \bigcup_{\forall_j} e_{ij}$.
- H_i a set of relations h_{ij} of type i that represent a hyperedge between a set of vertices such that $H_i = \bigcup_{\forall_j} h_{ij}$.

ScalA'17, November 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5125-6/17/11…$15.00
https://doi.org/10.1145/3148226.3148236

- $V_{h_{ij}}$ the set of vertices the hyperedge $h_{ij} \in H_i$ connects.
- P_i a set of pins which represent the connection from hyperedge h_{ij} to v where $h_{ij} \in H_i$ and $v \in V_{h_{ij}}$.

3 PARTITIONING

3.1 (Hyper)Graph partitioners

Graph-based partitioning methods are a common way to perform load balancing on relational data. These methods require the data to be transformed into a graph structure. The idea of this construction is such that the vertices represent a unit of work and the relations between the work form the edges of the graph. When this graph is partitioned the vertices are uniquely divided between the parts and the edges that cross the part boundaries represent communication between the connected parts. Graph partitioners split the graph into k parts where each part has the same amount of work and the inter part communication is minimized. Parallel multi-level techniques applied to graphs have been shown to produce high-quality partitions up to tens of thousands of parts quickly and efficiently [5, 17, 19, 22].

Hypergraph methods improve graph partitioning methods, using hyperedges to better represent more complex forms of communication. These hyperedges allow relations between multiple sets of work or graph vertices. In many cases, including unstructured meshes and sparse matrices, hypergraph partitioning has been shown to reduce the communication costs of the final partition [6, 7, 9]. While these methods produce better partitions of the data, they are more computational and memory intensive relative to graph-based methods.

3.2 Diffusive partitioning

Diffusive partitioning techniques perform improvements to existing partitions by migrating load across part boundaries. These migrations are either controlled globally, or locally computed on each part. The global methods select weight to migrate in order to minimize the total weight transferred or the maximum weight transferred in or out of a part [15, 20]. Local diffusive partitioners transfer load iteratively from parts with more work to neighboring parts with less [8, 24]. This approach combined with heuristics for how the load is transferred can have significantly less computational cost than global methods [11, 18].

3.3 The Current Contribution

EnGPar implements local diffusive partitioning for general usage on a range of applications with relational data structures. Towards this goal, an expanded graph structure, N-graph, is used to represent relational data structures from different applications. Then, diffusive partitioning techniques used within ParMA are implemented on the N-graph.

The re-implementation of the ParMA methods has been focused on array-based structures that can support efficient data parallel operations on GP-GPUs and vector units in many core processors. These changes will allow EnGPar to perform faster on next generation systems. In addition to the data parallel implementation of ParMA algorithms, certain portions of the algorithm have been improved to further increase the performance of EnGPar on all machines.

4 N-GRAPH

EnGPar interfaces to the different partitioning procedures through a multigraph abstraction called the N-graph. A multigraph [2, 4] is a graph that supports multiple edges between two vertices. Towards supporting a combination of (hyper)graph, geometric, and diffusive partitioning methods on relation-based data, the N-graph is defined using one of two modes. The first is a traditional multigraph with a set of vertices V and n different sets of edges $E_0, ..., E_{n-1}$. The N-graph also supports a multi-hypergraph mode utilizing hyperedges to better represent certain types of relations. This mode is defined by a set of vertices V, n sets of hyperedges $H_0, ...H_{n-1}$ and n sets of pins $P_0, ..., P_{n-1}$ which connect the vertices and hyperedges.

The hyperedge mode for the N-graph is well suited to relate vertices that share a single connection between greater than two vertices. Take for instance the construction of an N-graph given an unstructured mesh. Mesh elements, M^3, are represented by graph vertices and mesh vertices, M^0, are used for relations between mesh elements. Depending on the mode, edges/hyperedges and pins will be added to represent these relations. In the traditional graph mode, an edge is created between two graph vertices if the mesh elements they represent share a mesh vertex. In the hypergraph mode, one hyperedge is made for each mesh vertex. Also, a pin is created to connect a graph vertex and graph hyperedge if the corresponding mesh element is bounded by the mesh vertex.

To examine these two modes, let n be the number of mesh elements that bound a certain mesh vertex. On average in a three-dimensional tetrahedron mesh, n is 23 [3]. To represent this relation between all n graph vertices in the N-graph with traditional edges would require $O(n^2)$ edges in the N-graph. However, when using hyperedges one graph edge is created to represent the mesh vertex and $O(n)$ pins to connect the graph vertices and hyperedges. This results in a reduction in memory usage.

Figure 1 gives an example 2D mesh where seven mesh elements bound a mesh vertex (a). The N-graph of this mesh constructed using mesh faces as graph vertices and vertices for faces is shown in (b) using traditional edges and in (c) with hyperedges. In (b) fifteen graph edges are created for the one mesh vertex while in (c) one hyperedge is added and seven pins connect the graph vertices to the hyperedge.

The N-graph also supports having multiple edge types, whether using traditional edges or hyperedges. This allows representing multiple layers of connection between the graph vertices. This is useful to represent applications that use multidimensional data or complex levels of communication. One example is an unstructured mesh which have vertices, edges, and faces (in three dimensions) that are shared between mesh elements. To represent these data structures for a range of application needs, the N-graph supports the arbitrary use of edge types to allow different configurations for applications. Figure 2 depicts the mapping of a 2D unstructured mesh (a) to a representation where mesh elements map to graph vertices and mesh vertices map to graph hyperedges (b). In (c) a second mapping is shown where mesh edges are also mapped to a second edge type in the graph. The labels of the entities in (a) are carried to (b) and (c) to show which mesh entity is represented by the corresponding graph entity.

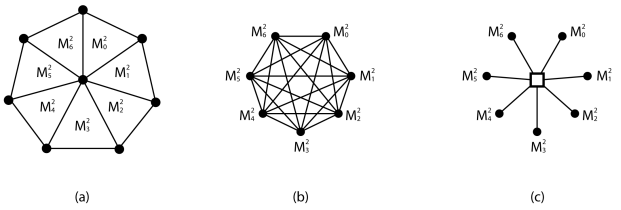

Figure 1: (a) a seven triangle mesh surrounding one vertex. (b) N-graph constructed around the vertex using traditional edges to connect the graph vertices whose corresponding mesh elements share the mesh vertex. (c) The N-graph construction using a hyperedge for the mesh vertex and connecting the adjacent mesh elements with pins in the N-graph.

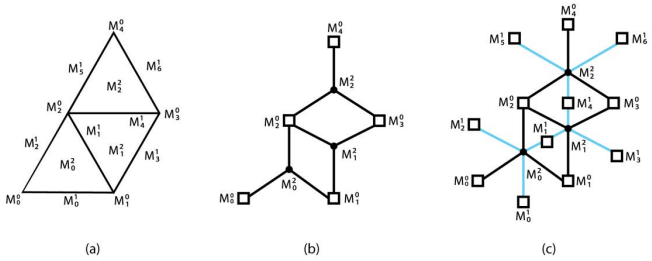

Figure 2: (a) a 2D unstructured mesh. (b) N-graph construction with elements→vertices, vertices→hyperedges. (c) additional mesh edges are used for a second set of hyperedges. Mesh labeling is shared in all three to correlate mesh entities to graph entities.

5 DIFFUSIVE LOAD BALANCING

The diffusive load balancing techniques used in both ParMA and EnGPar follow the same basic framework. The techniques iteratively perform a series of steps until user specified criteria are met or the algorithm can not improve the partition quality further. The main metric used in EnGPar for partition quality is the imbalance of a set of entities. This value is defined by calculating the sum of the weights of the entities in each part. Then, the maximum across all parts is divided by the average. For example, when equal weights are used, the imbalance of vertices in the N-graph is the maximum number of vertices on a single part divided by the average number of vertices per part, across all processes.

Algorithm 1 lists a general framework of the multi-criteria load balancing procedures in ParMA.

The framework's BALANCE procedure iteratively calls RUNSTEP to improve the target imbalance. The RUNSTEP procedure performs six stages to determine the diffusive load balancing of each step. The first step, *makeSides* on line 2 determines who are the neighbors of each part and what is the size of the part boundaries between them. The second calls *makeWeights*, which computes the weight of the target dimension d and sends the weight to the neighboring parts provided by the *sides* object. Then, the *makeTargets* function

Algorithm 1 ParMA Load Balancing Framework

```
1:  procedure RunStep(mesh,d)
2:      sides = makeSides(mesh)
3:      weights = makeWeights(mesh, sides, d)
4:      targets = makeTargets(mesh, sides, weights)
5:      queue = makeIterationQueue(mesh)
6:      plan = select(mesh, targets, queue)
7:      trim(mesh, plan)
8:      mesh.migrate(plan)
9:  end procedure
10: procedure Balance(mesh,dimensions)
11:     for all d ∈ dimensions do
12:         while imbalance of d > tolerance do RunStep(mesh,d)
13:             if Balancing Stagnates then
14:                 break
15:             end if
16:         end while
17:     end for
18: end procedure
```

determine which neighboring parts have less weight and determine how much weight to send to those parts. The fourth stage constructs an ordering of the entities on the boundary for selecting on line 7. A migration plan is then constructed on line 8 followed by trimming the plan on line 9. This trim operation ensures that the migration plan does not increase the imbalance of previously balanced dimensions. Finally, migration is performed to send the selected entities to neighboring parts on line 10.

The input to the BALANCE procedure, *dimensions*, is an ordering of the criteria to be balanced. It is interpreted such that earlier entries have higher priorities and thus will be balanced first and their reduced imbalance will be maintained in following steps. Line 8 of Algorithm 1 adjusts the migration plan to ensure that the imbalance of completed dimensions is not increased. An example of multi-criteria load balancing is the "vertex > element" case for finite element methods when degrees of freedom are defined by the mesh vertices. The degrees of freedom contain the largest portion of computational work in these simulations and thus, make balancing the mesh vertices a top priority. However, these simulations also require balancing the mesh elements for efficient linear system assembly. In this example mesh vertices would be the first criteria followed up by mesh elements.

The work to improve the multi-criteria load balancing in EnGPar is focused on two aspects of this algorithm. One is the generalization to support for a larger range of applications while the second is to improve the speed of the more communication and computational expensive parts. Each of these is discussed in more detail in the following sections.

5.1 Generalization of Multicriteria Load Balancing

The generalization of the framework in EnGPar to support more applications is partially completed with the usage of the N-graph. Since ParMA works directly on meshes, it cannot scale-free graphs [13, 21] or directly work with unstructured meshes using vertex-based partitions. Since EnGPar utilizes the N-graph, it natively supports other relation based data.

Beyond supporting other forms of data, many applications have different priorities of balancing the different criteria. While ParMA is built to perform load balancing targeting finite element applications, EnGPar takes a more general approach allowing users to define the ordering and tolerances of criteria. This is highlighted in Algorithm 1 on line 11 with the *dimensions* argument. This argument controls the ordering in which dimensions are balanced with earlier dimensions having higher priority. In ParMA these inputs would be defined by the balancer that is applied. Meaning that for ParMA to support additional applications new balancers need to be defined for the user to use. In EnGPar the users can use any ordering of priorities without any new development. This is done by providing a richer user interface with inputs to control all necessary components of the diffusive balancing procedure.

5.2 Graph Distance

One of the optimizations in ParMA targets decreasing surface area across parts by ordering the selection of elements to migrate based on their topological distance from the center of the part. Several challenges arise in the method due to the existence of disconnected components throughout the part. To get an optimal ordering it is important to offset the distances of the disconnected components such that shallower components get a higher priority. This leads to the shallower components being migrated first.

ParMA computes the graph distance using multiple breadth-first traversals; one traversal to find disjoint sets, another to locate the topological centers of each set, and another to compute the distance. EnGPar reduces the number of traversals from three to two. This is done by locating the disconnected components during the traversal to compute distances using a disjoint set data structure.

EnGPar's distance computation algorithm is listed in Algorithm 2. The general breadth-first traversal is on lines 1 to 5. This performs a traversal over the entire graph where every time a vertex is reached the input *kernel* is run. The algorithm begins by calling `COMPUTEDISTANCE`. On line 51, the initial traversal is used to find the topological center of all disconnected components simultaneously as well as the depth of each vertex. The first traversal is seeded by the vertices that are on the part boundary. Then, an additional traversal is performed on lines 55-60. This traversal is seeded by the deepest vertices of the first traversal found in the `SETLABELS` procedure on lines 25-39. This traversal is repeated until every vertex has a distance by computing the next deepest level of vertices that does not have a distance.

When multiple vertices share the same depth, it is possible that multiple disconnected components will be traversed simultaneously. To differentiate the components, a disjoint set is initialized for each vertex in the deepest level. The disjoint set data structure, commonly known for its use in Kruskal's algorithm for finding minimum spanning trees [12], allows fast joining and comparison of sets. The data structure consists of a label for each vertex and a parent pointer that initially points to itself. When two sets are unioned, one of the sets is chosen as the parent and the other set points its parent pointer to that set. This allows all sets to be represented by its highest parent allowing comparison of two sets in $O(\log n)$, in the average case, where n is the number of sets in the beginning. The remaining usage of the disjoint sets is in the `DISTANCE_KERNEL`

Algorithm 2 EnGPar Graph Distance Computation

```
1:  procedure TRAVERSAL(seed, graph, kernel)
2:      Breadth-first traversal of the graph vertices
3:      starting with the seed vertices.
4:      The kernel is called for each visited vertex.
5:  end procedure
6:  procedure DEPTH_KERNEL(u, G)
7:      for v ∈ e(u, v) do
8:          if not visited(v) then
9:              depth(v) ← depth(u) + 1
10:         end if
11:     end for
12: end procedure
13: procedure DISTANCE_KERNEL(u, G)
14:     for v ∈ e(u, v) do
15:         if not FIND(v) then
16:             label(v) ← FIND(u)
17:         else if FIND(u) != FIND(v) then
18:             UNION(u,v)                          ▷ merge them into the same set
19:         end if
20:         if not visited(v) then
21:             distance(v) ← distance(u) + 1
22:         end if
23:     end for
24: end procedure
25: procedure SETLABELS(visited,levels)
26:     init ← ∅
27:     for u ∈ levels do
28:         if not visited(u) then
29:             break
30:         end if
31:     end for
32:     maxdepth ← depth(u)
33:     for u ∈ levels do
34:         if not visited(u) and depth(u) == maxdepth then
35:             init ← init ⋃ u
36:         end if
37:     end for
38:     return init
39: end procedure
40: procedure MAXVISITEDDEPTH(visited)
41:     depth ← 0
42:     for u ∈ visited do
43:         if visited(u) and depth < depth(u) then
44:             depth ← depth(u)
45:         end if
46:     end for
47:     return depth
48: end procedure
49: procedure COMPUTEDISTANCE(G)
50:     init ← vertices classified on d − 1 partition model entities
51:     TRAVERSAL(init,G,depth_kernel)
52:     levels ← SORT(G)          ▷ levels is an array vertices in order of decreasing depth
53:     visited ← array(|V|, 0)
54:     depth ← 0
55:     while init ← SETLABELS(visited,levels) do   ▷ Label the vertices with
        the largest remaining depth. If none remain, then break from the while loop.
56:         foreach u ∈ init: MAKESET(u)
57:         foreach u ∈ init: depth(u) ← depth
58:         TRAVERSAL(init,G,distance_kernel)
59:         depth ← MAXVISITEDDEPTH(visited)
60:     end while
61: end procedure
```

on lines 15-19 where new vertices are assigned a label and already assigned vertices are unioned together. If there are multiple disjoint sets after the traversal is complete, the distances of each disjoint set is offset such that the deepest disjoint sets have lower distances.

6 EXPERIMENTS

EnGPar and ParMA are compared by running a workflow starting from a PUMI mesh and ending with a repartitioned PUMI mesh. For ParMA, this consists of running its load balancer on the PUMI mesh, but for EnGPar, this includes conversion from the PUMI mesh to the N-graph, running the EnGPar balancer, and migrating the PUMI mesh elements following the EnGPar partitioning assignments.

For our experiments, the N-graph is constructed such that the mesh elements are represented by graph vertices and the mesh vertices correspond to graph hyperedges. The final repartition of the PUMI mesh is done by running the PUMI migration routine used in ParMA with the partition of the N-graph.

We compare the performance of EnGPar's load balancing procedure to ParMA on the Mira BlueGene/Q system at the Argonne Leadership Computing Facility. The experiments use one processes per hardware thread [14].

Tests were run on a one billion element mesh of an airplane's vertical tail structure. The original 4Ki($4 * 2^{10}$) part mesh was partitioned to 8Ki parts using global ParMETIS part k-way [17] which repartitions the global structure to ensure equal load across all parts. Then, partitions in powers of two are created from 128Ki to 512Ki parts using local ParMETIS part k-way which splits each part, but only balances the newly split parts instead of the global partition. The local method is much faster than the global method especially at large part counts. These meshes start with an element imbalance of 1.02 while the vertex imbalance ranges from 1.12 at 128Ki to 1.53 512Ki processes. ParMA and EnGPar are run with mesh vertices > mesh elements criteria with a target imbalance of 1.05.

Statistics are gathered for the imbalance of vertices and elements. Additionally, since PUMI copies mesh vertices that exist on part boundaries we report the average number of mesh vertices per part before and after EnGPar and ParMA are run. Lastly, the overall time of the balancing is provided. In the case of EnGPar, this time includes the conversion from PUMI mesh to N-graph and the repartition of the mesh after running EnGPar's balancer. Finer grain timing is reported for a deeper understanding of the most expensive operations in ParMA and EnGPar.

7 RESULTS

Figure 3 shows the vertex imbalance and element imbalance respectively. The original partition, labeled 'initial' in the charts, is also shown as well as a line representing the target imbalance. For vertices, EnGPar is able to reduce the imbalance to the target level in both the 128Ki and 256Ki cases. In the 512Ki case EnGPar significantly reduces the imbalance from 1.53 to 1.12, but it is not able to reach the target imbalance like ParMA does. EnGPar is able to maintain the element imbalance at the target level for all three process counts.

Table 1 details the average number of vertices in each part of the mesh. As larger surface area leads to more shared vertices across part boundaries in the PUMI mesh, the average number of shared

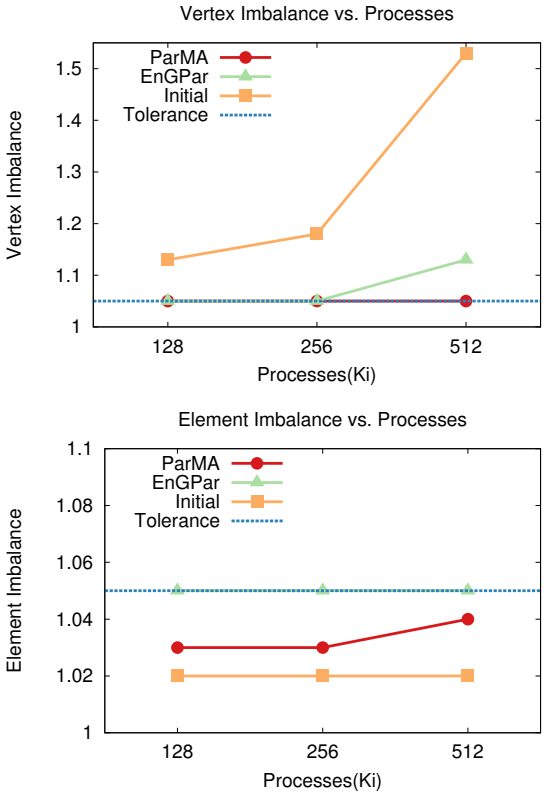

Figure 3: Vertex(top) and Element(bottom) imbalances for the initial partitioning and the partitions from EnGPar and ParMA. Additionally a line representing the tolerance is provided.

vertices increases. This increase results in more computational work and communication for the application. ParMA maintains the average number of vertices with a slight decrease in all cases. EnGPar increases the number of vertices by one to four percent across the three runs. EnGPar currently does not target surface area as a criterion and therefore will prioritize the entity imbalance over an increase in surface area.

	128Ki	256Ki	512Ki
Initial	2146.404	1138.881	611.673
ParMA	2141.965	1137.343	610.959
EnGPar	2161.521	1155.727	637.354

Table 1: Average number of mesh vertices per part.

Figure 4 shows the runtime for ParMA and EnGPar. Since EnGPar does not reach the tolerance in the 512Ki parts case, the timing presented for ParMA is for a run where the target imbalance is set to the imbalance that EnGPar reaches when it finishes, 1.12 vertex and 1.05 element imbalance. In all cases, EnGPar runs faster than ParMA. For the 128Ki and 256Ki cases, where EnGPar successfully reaches the target imbalance, EnGPar is around 49% and 38% faster than

ParMA. In the 512Ki case, EnGPar reaches the 1.12 vertex imbalance and 1.05 element imbalance targets 33% faster than ParMA does.

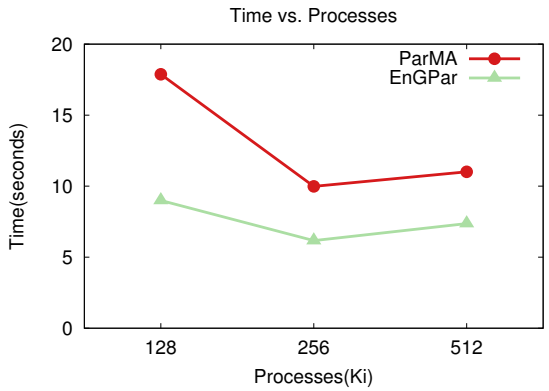

Figure 4: Runtime of ParMA and EnGPar running a vertex>elements balancer at 128Ki, 256Ki, and 512Ki processes. ParMA is run until the mesh has been balanced to the end imbalance that EnGPar reaches.

Figures 5 and 6 displays the time spent computing the graph distance and migrating entities respectively. We report these two operations because they have the most computational work in ParMA. Figure 5 shows the improvements in EnGPar's graph distance computation algorithm over ParMA's with an 80-90% speedup across each part count. Also, EnGPar continues to scale across all three runs, while ParMA runs slower at 512Ki than 256Ki. The improvements to the graph distance computation are responsible for most of the speedup to the EnGPar balancing over ParMA.

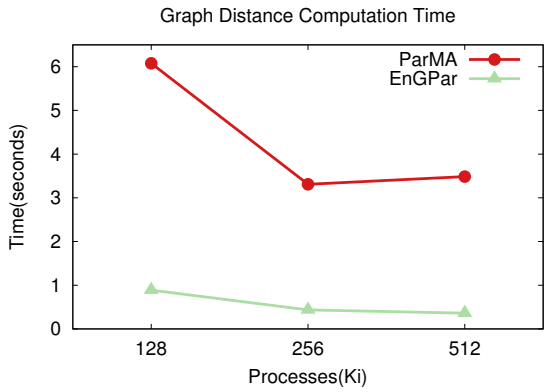

Figure 5: Time spent computing the graph distance for ParMA and EnGPar at 128Ki, 256Ki, and 512Ki processes.

While there is a large speedup because of the graph distance computation, the same is not the case for migration. As seen in Figure 6, ParMA spends less time in migration than EnGPar for all three cases. At 128Ki ParMA spends 9% less time, 16% at 256Ki and 32% at 512Ki. Even though EnGPar is spending more total time migrating entities than ParMA, the EnGPar migration routine generally runs faster than ParMA's due to having less data to migrate. In Table 2 the number of iterations of the RUNSTEP function in Algorithm 1 is shown for both ParMA and EnGPar. In all three cases, EnGPar takes a few more iterations than ParMA to reach the target imbalance. Thus, EnGPar performs more migrations of entities than ParMA. This results in an increase in the runtime of the load balancer.

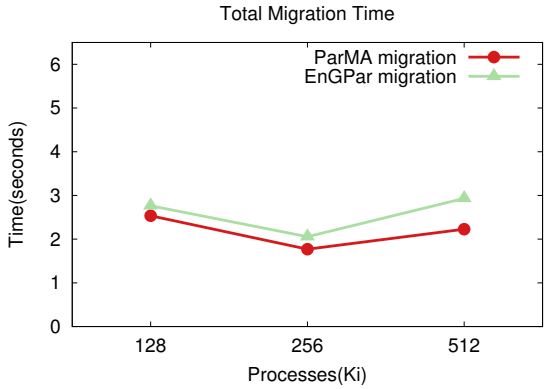

Figure 6: Time spent migrating entities in ParMA and EnGPar at 128Ki, 256Ki, and 512Ki processes.

	128Ki	256Ki	512Ki
ParMA	7	6	10
EnGPar	8	9	16

Table 2: Number of iterations of the RunStep procedure from Algorithm 1 for each process count for EnGPar and ParMA.

Figure 7 breaks down the three portions of the timing for the EnGPar runs. The construction of the N-graph exhibits strong scaling quite nicely, however repartitioning scales much slower. This step is dominated by the PUMI migration, so the step's scaling is driven by the scaling of the user's migration routine. Unlike construction and repartition, balancing does not decrease in runtime with increasing number of parts. The amount of time that is spent migrating entities during balancing is shown as the bottom portion of each bar. Migration takes from 49% to 51% of the balancing step of each run.

8 CLOSING REMARKS AND FUTURE WORK

EnGPar has been shown to significantly reduce, or maintain, the imbalance for multiple criteria simultaneously. EnGPar is able to perform these load balancing algorithms faster than its predecessor ParMA. However, there are cases where EnGPar stagnates and is unable to reach the target imbalance that ParMA is able to achieve. Furthermore, there is an increase in the number of mesh vertices in the final partition which means that the length of the part boundaries is increasing. Thus, we are tracking down the causes of these deficiencies and how to resolve them.

One deficiency we have tracked is due to the surface area of parts. This is a result of one aspect of ParMA that is currently

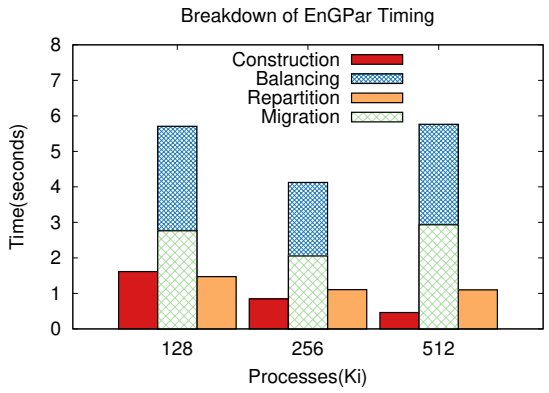

Figure 7: A breakdown of timing for each step of the EnGPar runs. The bottom part of the bar for balancing represents the migration component of the balancer.

not in EnGPar. In ParMA, several stages of element selection are affected by the current surface area of the part and the effect of migration on the surface area. In EnGPar, these checks have not been implemented and is likely a major cause of the increase in the number of vertices in the mesh.

While EnGPar is running faster than ParMA, there are several inefficiencies that improving upon would further speedup EnGPar. Most of the inefficiencies are caused by the increase in the number of iterations it takes to finish. Sending more weight in each step or improving the selection of entities to send will lead to fewer iterations required and thus faster runtimes. Beyond just the number of iterations, many of the algorithms in EnGPar can be performed well on GPUs. Implementing the algorithms on GPUs will lead to further speedups in key portions such as the graph distance computation and migration. Towards this, initial implementations of the breadth-first (hyper)graph traversal is being written and tested using Kokkos [10].

The results here are all given for unstructured meshes and compared to EnGPar's predecessor, ParMA. Now that EnGPar has the ability to perform load balancing operations on general structures rather than just meshes, these techniques can be applied to other applications that use relation based data structures like scale-free graphs and vertex-based meshes.

ACKNOWLEDGMENTS

This research is supported by the National Science Foundation under Grant ACI 1533581. The support of the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under award DE-SC00066117 (FASTMath SciDAC Institute) is also acknowledged.

An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program and a separate award of computer time by the Theta Early Science program. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

REFERENCES

[1] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. 1999. Achieving High Sustained Performance in an Unstructured Mesh CFD Application. In *Proc. Int. Conf. High Performance Comput., Networking, Storage and Anal. (SC)*. IEEE, Article 69, 13 pages. https://doi.org/10.1145/331532.331600

[2] Jorgen Bang-Jensen and Gregory Gutin. 1997. Alternating cycles and paths in edge-coloured multigraphs: A survey. *Discrete Math.* 165, 1 (March 1997), 39–60. https://doi.org/10.1016/S0012-365X(96)00160-4

[3] Mark W. Beall and Mark S. Shephard. 1997. A General Topology-Based Mesh Data Structure. *Int. J. Numerical Methods in Eng.* 40, 9 (May 1997), 1573–1596. https://doi.org/10.1002/(SICI)1097-0207(19970515)40:9<1573::AID-NME128>3.0.CO;2-9

[4] Frank Boesch and Ralph Tindell. 1980. Robbins's Theorem for Mixed Multigraphs. *The Amer. Math. Monthly* 87, 9 (Nov. 1980), 716–719.

[5] UV Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar. 2013. UMPa: A multiobjective, multi-level partitioner for communication minimization. *Contemporary Math.* 588, 1 (Feb. 2013), 53–64.

[6] Umit V Çatalyürek and Cevdet Aykanat. 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *Parallel and Distributed Syst., IEEE Trans.* 10, 7 (July 1999), 673–693.

[7] Umit V Çatalyürek, Erik G Boman, Karen D Devine, Doruk Bozdağ, Robert T Heaphy, and Lee Ann Riesen. 2009. A repartitioning hypergraph model for dynamic load balancing. *J. Parallel and Distributed Comput.* 69, 8 (Aug. 2009), 711–724.

[8] George Cybenko. 1989. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel and Distributed Comput.* 7, 2 (Oct. 1989), 279–301.

[9] Karen D Devine, Erik G Boman, Robert T Heaphy, Rob H Bisseling, and Umit V Çatalyürek. 2006. Parallel hypergraph partitioning for scientific computing. In *Parallel and Distributed Process. Symp., 2006. IPDPS 2006. 20th Int.* 10–pp.

[10] H. C. Edwards and C. R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In *2013 Extreme Scaling Workshop*. 18–24. https://doi.org/10.1109/XSW.2013.7

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

[12] Zvi Galil and Giuseppe F. Italiano. 1991. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comput. Surv.* 23, 3 (Sept. 1991), 319–344. https://doi.org/10.1145/116873.116878

[13] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.. In *OSDI*, Vol. 12. 2.

[14] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, a. gara, G. Chiu, P. Boyle, N. Chist, and C. Kim. 2012. The IBM Blue Gene/Q Compute Chip. *IEEE Micro* 32, 2 (March 2012), 48–60. https://doi.org/10.1109/MM.2011.108

[15] YF Hu and RJ Blake. 1999. An improved diffusion algorithm for dynamic load balancing. *Parallel Comput.* 25, 4 (April 1999), 417–444.

[16] Daniel A. Ibanez, E. Seegyoung Seol, Cameron W. Smith, and Mark S. Shephard. 2016. PUMI: Parallel Unstructured Mesh Infrastructure. *ACM Trans. Math. Softw.* 42, 3, Article 17 (May 2016), 28 pages. https://doi.org/10.1145/2814935

[17] George Karypis and Vipin Kumar. 1999. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Rev.* 41, 2 (June 1999), 278–300.

[18] Brian W. Kernighan and S. Lin. 1970. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Tech. J.* 49, 2 (April 1970), 291–307.

[19] Dominique LaSalle and George Karypis. 2013. Multi-threaded graph partitioning. In *Parallel & Distributed Process. (IPDPS), IEEE 27th Int. Symp.* 225–236.

[20] Chao-Wei Ou and Sanjay Ranka. 1997. Parallel Incremental Graph Partitioning. *IEEE Trans. Parallel Distrib. Syst.* 8, 8 (Aug. 1997), 884–896. https://doi.org/10.1109/71.605773

[21] Robert S Pienta and Richard M Fujimoto. 2013. On the parallel simulation of scale-free networks. In *Proc. 2013 ACM SIGSIM Conf. Principles of advanced discrete simulation*. 179–188.

[22] Kirk Schloegel, George Karypis, and Vipin Kumar. 2002. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience* 14, 3 (March 2002), 219–240.

[23] Cameron W. Smith, Michel Rasquin, Dan Ibanez, Kenneth E. Jansen, and Mark S. Shephard. [n. d.]. Improving Unstructured Mesh Partitions for Multiple Criteria Using Mesh Adjacencies. *SIAM J. Scientific Comput.* ([n. d.]). submitted for publication.

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

[25] M. Zhou, O. Sahni, K.D. Devine, M.S. Shephard, and K.E. Jansen. 2010. Controlling unstructured mesh partitions for massively parallel simulations. *SIAM J. Scientific Comput.* 32, 6 (Nov. 2010), 3201–3227.

[26] Min Zhou, Onkar Sahni, Ting Xie, Mark S Shephard, and Kenneth E Jansen. 2012. Unstructured mesh partition improvement for implicit finite element at extreme scale. *The J. Supercomputing* 59, 3 (Dec. 2012), 1218–1228.