Engineering with Computers (2012) 28:287–301
DOI 10.1007/s00366-011-0218-x

ORIGINAL ARTICLE

Tools to support mesh adaptation on massively parallel computers

Min Zhou · Ting Xie · Seegyoung Seol · Mark S. Shephard · Onkar Sahni · Kenneth E. Jansen

Received: 27 January 2010 / Accepted: 28 March 2011 / Published online: 21 April 2011

Abstract The scalable execution of parallel adaptive analyses requires the application of dynamic load balancing to repartition the mesh into a set of parts with balanced work load and minimal communication. As the adaptive meshes being generated reach billions of elements and the analyses are performed on massively parallel computers with 100,000's of computing cores, a number of complexities arise that need to be addressed. This paper presents procedures developed to deal with two of them. The first is a procedure to support multiple parts per processor which is used as the mesh increases in size and it is desirable to partition the mesh to a larger number of computing cores than are currently being used. The second is a predictive load balancing method that sets entity weights before dynamic load balancing steps so that the mesh is well balanced after the mesh adaptation step thus avoiding excessive memory spikes that would otherwise occur during mesh adaptation.

Keywords Mesh adaptation · Parallel computation · Dynamic load balancing

M. Zhou · T. Xie · S. Seol · M. S. Shephard (✉) · O. Sahni
Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY 12180, USA
e-mail: shephard@scorec.rpi.edu

M. Zhou
e-mail: zhoum@scorec.rpi.edu

T. Xie
e-mail: txie@scorec.rpi.edu

S. Seol
e-mail: seol@scorec.rpi.edu

O. Sahni
e-mail: osahni@scorec.rpi.edu

K. E. Jansen
Department of Aerospace Engineering Sciences, University of Colorado at Boulder, 429 UCB, Boulder, CO 80309, USA
e-mail: Kenneth.Jansen@Colorado.edu

1 Introduction

Unstructured mesh methods, like finite elements and finite volumes, support the effective analysis of complex physical behaviors modeled by partial differential equations over general three-dimensional domains. The most reliable, and potentially efficient, application of unstructured mesh methods is through the application of adaptive methods where a posteriori error estimates, or indicators, are used to determine correction indicators used to control where and how the mesh is to be modified. In the case of unstructured meshes local mesh modification techniques can be applied to automatically perform the required mesh modifications.

Although well-defined adaptive meshes can have two to three orders of magnitude fewer elements than a more uniform mesh for the same level of accuracy, there are many complex simulations where the meshes required are so large that they can only be solved on massively parallel computers. The execution of a simulation on a massively parallel computer requires the mesh be distributed over the computing nodes and/or cores of the parallel computer. Although there are different strategies for the distribution of the mesh on the parallel computer, the most common approach is to partition the mesh into a number of parts in which the amount of computation required for each part is nearly equal and the amount of inter-processor communication between parts required during the computation is as small as possible. Section 2 provides more specific

information on the definition of a mesh partition as a collection of parts. For purposes of these introductory remarks, the following factors are of importance:

- Understanding how the mesh entities and their adjacencies relate to computation and/or memory use.
- Adaptive mesh modifications will change the number of mesh entities and the mesh adjacencies, thus changing the load balance of the mesh between the parts of the partition. Thus, dynamic load balancing must be applied to regain load balance.
- The efforts involved with setting-up and constructing a partition of a mesh is substantial enough that it must be performed efficiently.
- The meshes used in massively parallel simulations are so large that they cannot be managed using the memory of a single core or a single node.

The key consequence of the above factors is that parallel adaptive simulation on massively parallel computers requires all steps be efficiently executed in parallel on a partitioned mesh and that the mesh be effectively repartitioned as needed at various steps in the simulation.

Although a reasonable set of procedures and methods had been developed for parallel mesh adaptation when a few hundred processors were used [1, 2], recent efforts to move to massively parallel computers with 100,000's of compute cores [3–5] indicated the need to develop a number of additional capabilities [6]. This paper focuses on two of these capabilities.

The first capability is an extension of the partition model to support multiple parts per processor. Although we have run into multiple scenarios where this capability is useful, the overriding one in massively parallel adaptive simulations is that the size of the mesh often grows by one or two orders of magnitude during a simulation, yielding a situation where it is necessary to increase the number of cores being used in subsequent analysis steps.

The second capability is a predictive load balancing procedure that is used to drive a repartitioning of the mesh before a mesh adaptation step with the goal of having nearly a balanced mesh partition after the mesh is adapted. Although this capability is potentially useful in maintaining computational load balance during mesh adaptation, its primary goal is to control the amount of memory used by each part so that memory usage is nearly equal with no parts requiring substantially more than the average memory usage. On some of the most effective massively parallel computers, exceeding the memory on a single core will lead to failure or termination of the calculation.

The next section introduces the basic tools and methods used to partition unstructured meshes on distributed memory parallel computers. Section 3 summarizes the distributed mesh representation used and its extension to support multiple parts per processor. The predictive load balancing algorithm is described in Sect. 4.

2 Partitioning of unstructured meshes

Massively parallel computers are characterized by having a large number of compute nodes where each node will typically include a number of computing cores (currently on the order of 16, with this number increasing substantially on future machines). The memory is configured with either a core level or node level address space. In either case, when compared to the combined memory of the entire parallel computer, the amount of memory within a single address space is small. Therefore, at any point in the parallel computation, the mesh is housed over multiple memory spaces and the ability to perform computations on the mesh, including adapting the mesh, must maintain an understanding of the mesh entities and how they interact across these multiple memories.

For the purpose of the current paper, it will be assumed that the basic unit of computation will be the individual computing core. It is further assumed that all interactions between cores will be controlled through message passing interface (MPI) calls where MPI is well known to be an effective means to program distributed memory parallel computers.

Unstructured meshes are a decomposition of the domain into a set of topologically simple geometric entities (e.g., triangles and quadrilaterals in 2D, tetrahedral, wedges, pyramids, hexahedron, etc. in 3D) where the patterns of how the entities are connected follow basic rules on the matching of the boundaries of the entities (e.g., edge to edge and face to face), but are otherwise unstructured (e.g., number of edges coming into a vertex, number of faces using an edge, etc. varies over the mesh). Unstructured meshes are most effectively described using a topology-based representation in which the members of the hierarchy of topological entities of regions, faces, edges and vertices are defined and how they are connected is stored. There are a number of options as to which entities are explicitly represented and which connectivities, in terms of entity adjacencies, are stored [7–10].

Since any reasonable implementation of a mesh representation will only maintain a subset of the 12 possible entity adjacencies, a key concept of importance in the definition of a mesh representation is the level of complexity required to determine any adjacency that is not explicitly stored. Of specific importance to the current paper is the use of a method that employs a complete representation where the definition of a complete representation is one for which any requested mesh adjacency can be determined in O(1) time, meaning its determination

will not require a traversal of data that is a function of the size of the mesh [10]. The efficient implementation of any procedure that deals with the general modification of the mesh will require a complete representation be used. The following nomenclature provides an effective means to describe mesh entities and their adjacencies.

Mesh nomenclature

M	an abstract model of the mesh.
$\{M\{M^d\}\}$	a set of topological entities of dimension d in model M.
M_i^d	the i^{th} entity of dimension d in model M. $d = 0$ for a vertex, $d = 1$ for an edge, $d = 2$ for a face, and $d = 3$ for a region.
$\{M_i^d\{M^q\}\}$	a set of entities of dimension q in model M that are adjacent to M_i^d.
$M_i^d\{M^q\}_j$	the j^{th} entity in the set of entities of dimension q in model M that are adjacent to M_i^d.

Examples

$\{M\{M^2\}\}$	the set of all the faces in the mesh.
$\{M_3^1\{M^3\}\}$	the mesh regions adjacent to mesh edge M_3^1.
$M_1^3\{M^1\}_2$	the second edge adjacent to mesh region M_1^3.

In the cases of interest the unstructured mesh must be distributed over the memories of a large number of cores, thus methods are needed to decide which mesh entities will be on each core and how the mesh adjacency information between neighboring mesh entities is maintained for the entire mesh, including maintaining those relationships when the mesh is adapted. One common approach for the distribution of the mesh is to partition it into a set of parts where each part is a set of mesh entities, with the goal of having reasonably continuously connected mesh entities within a single part such that the number of mesh entities on the inter-part boundaries, those mesh entities shared between two or more parts, is minimized in some reasonable fashion. Following somewhat standard practices, and more specifically those being developed by the DOE SciDAC for Interoperable Technologies for Advanced Petascale Simulations [11, 12], the following are key requirements placed on the definition of mesh parts in a partition:

- The mesh entities on a single part will maintain the required on-part adjacencies and those mesh entities can be operated on in serial.
- Parallel methods are provided to be able to determine inter-part adjacency information for the mesh entities (faces, edges and vertices) on the inter-part boundaries.
- Parallel methods are provided to move mesh entities between parts that include constructing/updating the required on-part and inter-part mesh adjacencies.
- The mesh entities associated with a single part will be in the memory of a single core.
- Multiple parts can be maintained on a single core.
- The union of the mesh entities over all the parts in the partition defines the complete mesh being operated on in parallel.

One further requirement being used with the methods and tools employed in this paper is that the mesh representation being used is complete at the level of the part and the entire mesh. From a practical perspective, maintaining a complete representation such that any needed adjacency can be found without global traversals must be required of any parallel tools that execute mesh adaptation, otherwise there is no chance to gain parallel efficiency or scalability.

To support the full set of mesh adjacencies, the mesh entities on inter-part boundaries are duplicated on the parts where they are used in adjacency relations. Mesh entities that are not on any inter-part boundary exist on a single part only. Figure 1 depicts a mesh that is distributed on three parts. Vertex M_1^0 is common to three parts and exists on each part, several mesh edges like M_j^1 are common to two parts. The dashed lines represent inter-part boundaries that consist of mesh vertices and edges (including mesh faces in 3D) duplicated on multiple parts.

Since the execution of an on-part adjacency request will only return those entities on that part, a parallel device, referred to as the partition model [10], is needed to provide the information to indicate when there are adjacent entities on other parts that must be considered and provide the ability to access those adjacencies. The partition model introduces a set of topological entities that represent the collections of mesh entities that define them with respect to

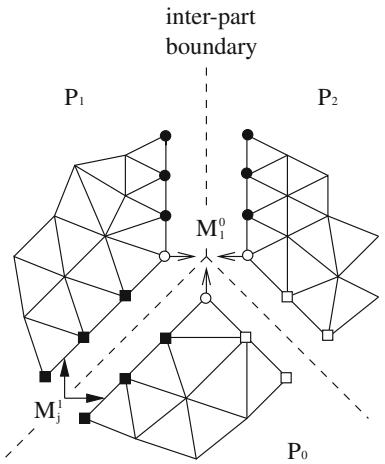

Fig. 1 Distributed mesh on three parts P_0, P_1 and P_2 [10]

the partition model. The key construct used to support determining if there are off part adjacent mesh entities is the concept of the classification of the mesh entities with respect to the partition model. Before being able to define the classification of mesh entities against partition model entities a device is needed to uniquely define those entities. Following [10] this device is the residence part operator:

Definition 1 (*Residence part operator* $\mathscr{P}[M_i^d]$) An operator that returns a set of part ID(s) where M_i^d exists.

If $\{M_i^d\{M^q\}\} = \emptyset$, $d < q$, the mesh entity is not bounded by any higher order mesh entities. In this case $\mathscr{P}[M_i^d] = \{p_i\}$ where p_i is the ID of the single part on which M_i^d exists. Otherwise, $\mathscr{P}[M_i^d] = \cup\mathscr{P}[M_j^q \mid M_i^d \in \{\partial(M_j^q)\}]$ which indicates the set of part ID's for mesh entities that M_i^d bounds.

Definition 2 (*Partition model entity*) A topological entity in the partition model, P_i^d, is defined as the group of mesh entities of dimension d that have the same $\mathscr{P}$. Each partition model entity is uniquely determined by $\mathscr{P}$.

Each partition model entity stores dimension, ID, residence part(s), and the owning part. From a mesh entity level, by keeping the proper relation to the partition model entity, all needed services to represent mesh partitioning and support inter-part communications can be supported.

Definition 3 (*Partition model classification*) The unique association of mesh topological entities of dimension $d_i, M_i^{d_i}$, to the topological entity of the partition model entity of dimension $d_j, P_j^{d_j}$ where $d_i \leq d_j$, on which it lies is termed partition classification and is denoted $M_i^{d_i} \sqsubset P_j^{d_j}$.

Definition 4 (*Reverse partition classification*) For each partition model entity, the set of equal order mesh entities classified on that entity defines the reverse partition classification. The reverse partition classification is denoted as $\mathrm{RC}(P_j^d) = \{M_i^d \mid M_i^d \sqsubset P_j^d\}$.

Figure 2 illustrates a distributed 3D mesh with mesh entities labeled with arrows indicating the partition classification of the entities onto the partition model entities and its associated partition model. The mesh vertices and edges on the thick black lines are classified on partition edge P_1^1 and they are duplicated on three parts P_0, P_1, and P_2. The mesh vertices, edges and faces on the shaded planes are duplicated on two parts and they are classified on the partition model face pointed with each arrow. The remaining mesh entities are not duplicated, therefore they are classified on the corresponding partition model region (i.e., part). Note the reverse classification returns only mesh entities of the same order as the partition entity they are classified on. The reverse classification of P_1^1 returns mesh edges on the thick black lines. The reverse classification of partition face $P^2{}_i$ returns mesh faces on each corresponding shaded plane, $i = 1, 2, 3$.

With the concepts of partition classification and/or reverse partition classification, combined with the on part adjacency information it is clear that the information required to construct complete adjacency information, accounting for the mesh over all parts, is available.

A number of algorithmic approaches have been developed to define a good mesh partition. For unstructured meshes, graph/hypergraph-based algorithms are by far the most effective and powerful. These methods model the computation and communication requirements as a graph or hypergraph, and then typically apply multilevel heuristic strategies to divide the graph/hypergraph into equally weighted subgraphs/hypergraphs [13–18] which will define the individual parts. The computation and communication costs of an operation on a mesh can be represented by a weighted graph $G(V, E)$. Specifically, nodes V of the graph represent mesh entities to be partitioned; each graph node may have a weight proportional to the computational cost for the associated entity. Graph edges E represent dependencies between two nodes. The graph edges are defined by the mesh adjacencies that indicate a dependency. For example defining a graph edge between the pair of mesh regions in the adjacency $\{M_i^2\{M^3\}\}$ would indicate a dependency based on the fact that they share the face M_i^2. Edges may also be weighted with the strength of the dependency in terms of amount of inter-part communication required.

The goal of graph partitioning is to create a partition of a graph that balances work loads among parts while minimizing inter-part communication. A partition of a graph $G(V, E)$ is a division of the nodes V of $G(V, E)$ into subsets or parts V_i such that $V = \bigcup_{i=1}^{k} V_i$ and $V_i \cap V_j = \emptyset$, $\forall$ $i \neq j$. Thus, graph partitioning identifies parts V_i with approximately equal total nodal weight [i.e., $\sum_{v \in V_i} \mathrm{weight}(v) \leq \frac{\mathrm{tol}}{k} \sum_{v \in V} \mathrm{weight}(v)$ for some tolerance (tol) while attempting to minimize the total weight of edges having endpoints in more than one part].

In the 3D unstructured meshes considered here the graph nodes are mesh regions or entity groups (a group of mesh regions the application indicated must be on the same part). While it is possible to create the set of graph edges associated with a full set of adjacencies, testing indicated that creating all the graph edges based on a full set of adjacencies requires substantially longer partitioning times without producing a better partition. Therefore, a subset of graph edges defined by only face adjacencies is used in the current study. It is important to note that although the graph

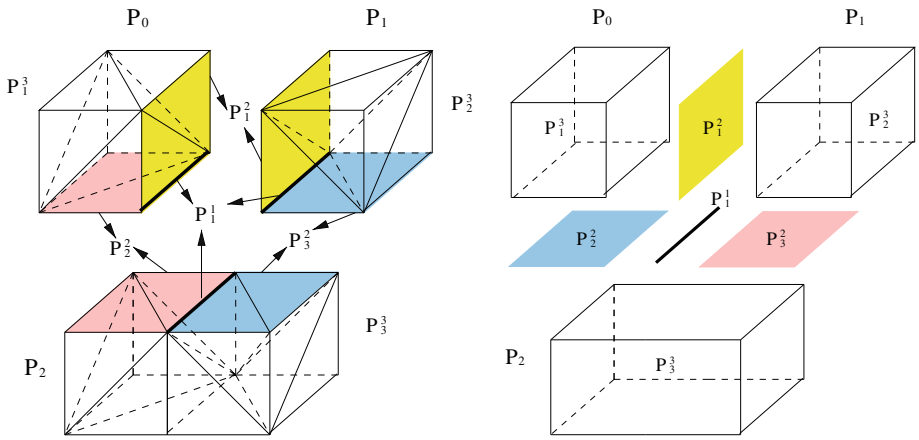

Fig. 2 Distributed mesh and its association with the partition model via classification

used by the partitioning algorithm employs only a subset of the mesh adjacencies, the parallel mesh data structure is still required to maintain all adjacencies used by the complete mesh representation being used.

Two types of graph edges are defined as follow. The edges connecting graph nodes on different processors that cross processor-boundaries, are referred to as inter-edges (i.e., inter-processor edges). On the other hand, the ones connecting graph nodes on the same processor that do not cross processor-boundaries, are referred to as intra-edges (i.e., intra-processor edges). Figure 3 is one example showing a mesh and the corresponding graph, in two dimensions for clarity. The mesh faces in 2D (regions in 3D) are defined as the graph nodes and the graph edges are defined for each of the graph nodes that share an edge (face in 3D). The mesh is partitioned into three parts and distributed on three processors (see Fig. 3a). The corresponding graph is built up for this mesh, which is shown in Fig. 3b. The solid lines demonstrate the intra-edges and the dashed ones represent the inter-edges.

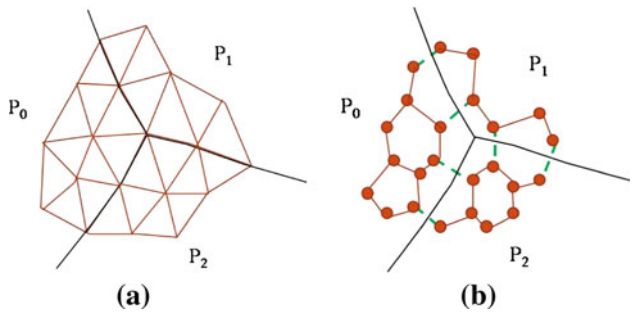

Fig. 3 A distributed mesh and its corresponding partition graph. **a** The mesh is partitioned into three parts and distributed on three processors (i.e., one part per processor). **b** Corresponding part graph. The *circles* represent the graph nodes. The *solid edges* demonstrate the intra-processor graph edges and the dashed ones demonstrate the inter-processor graph edges

3 Distributed mesh supporting multiple parts per processor

The key function performed during parallel mesh adaptation is moving selected mesh entities from one part to another part while making sure that the adjacency relations are properly maintained. The process, referred to as mesh migration, involves the basic steps of:

1. Collecting the mesh entities, and needed adjacencies, to be sent from one part to another and packing them for transfer.
2. Transferring the mesh entities to the new part.
3. Updating the local mesh adjacencies on the affected parts.
4. Updating the partition model and inter-part adjacency information.

The technical aspects of doing this process efficiently in parallel for the one part per processor are given in [10]. This section indicates the extensions required to support multiple parts per processor core which are focused primarily on the mesh migration process.

The mesh partition is a collection of mesh parts, P_i, distributed over the processor cores of the parallel computer. Each P_i consists of the set of mesh entities that are assigned to that part. Each P_i is assigned to a single processor core, C_j. In the case of single part per processor, there is a one-to-one correspondence between the part and processor. In the case of multiple parts per processor, each processor, C_j can have a set of parts assigned to it. In the case of multiple parts per processor it is necessary to be able to uniquely identify the part at any point in the process. One choice for this identifier is a global part ID, P_i which several of the applications that use the distributed mesh functions require. In fact, some applications require the ID be a continuous sequence (e.g., 0,1,2,…, *num-parts* − 1). In the cases we have seen to date, those applications that require unique single ID are ones where

the number of parts are static and the mesh in each part is fixed.

The use of a global part ID, P_i, is not ideal for the important application of adaptive mesh modification that includes the creation of multiple parts per processor and movement of mesh entities between parts, and movement of parts between processors. In these cases the alternative of the pair of local (to the processor) part identifier, LP_i, and the processor identifier C_j is more effective since it supports creating and/or eliminating parts without the need for global communication or algorithms to be sure the labeling meets a specific labeling convention. Thus, the multiple parts per processor partition model procedures support this pair level labeling. The same underlying strategy is used in identifying mesh entities where it is the combination of a local label, part local label and part ID's that uniquely identifies the mesh entities. This approach effectively supports the ability to deal with modifying the mesh and/or its partition effectively in parallel. To support interfacing to those applications that require a specific global ID, a set of procedures are included that will assign a unique set of single part ID's to each part upon request.

Figure 4 is an example of a partition with multiple parts per processor. The solid lines are the inter-processor part boundaries, while the dashed lines are the intra-processor part boundaries. Note the figure shows both a local and global part ID for each part.

One fairly obvious extension to support multiple parts per processor relates to iterating over all the parts in the partition. The single part per processor only needed to iterate over the processors. In the multiple parts per processor implementation, a loop over the number of parts per processor is inserted inside the loop over the processors.

Given the above approach to the implementation of multiple parts per processor, there is no need to alter any of the functions given in [10] that support migrating groups of mesh entities from one part to another. However, it is desirable to add a full part migration function which can efficiently move the entire definition of a part from one processor to another. This function can be made more efficient than the general mesh entity migration since all the mesh adjacencies remains the same and the only data that has to be changed on neighboring parts is for the partition classification and/or reverse partition classification to return the new LP_i, C_j for the part.

Using the capability to handle multiple parts per processor, two partitioning categories are defined based on data provided to the graph/hypergraph-based partitioner: *global partitioning* which considers both intra-processor and inter-processor edges, and *local partitioning* which considers only intra-processor edges. The definition is slightly different from the ones in [19], but they are essentially equivalent. Global partitioning considers all the graph edges and nodes, and provides a balanced partition with the total minimum inter-part communication. On the other hand, local partitioning considers only the on-processor (intra) graph edges and nodes, without knowing the existence of graph nodes and graph edges on other processors. In this case, partitioning is carried out independently on each processor, as a serial process, which can be trivially executed in parallel on all processors. The redistribution of mesh in Fig. 3 from three parts to six parts can be performed by these two categories of partitioning algorithm, respectively, with the capability of supporting multiple parts (two parts in this case) per processor. The resulting partitions are shown in Fig. 5a and b. The black solid lines represent the inter-processor part boundaries based on the global partition in Fig. 5a and local

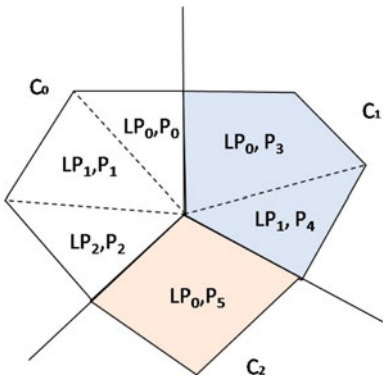

Fig. 4 An example of multiple parts per processor. The *solid lines* represent the inter-processor part boundaries and the *dashed lines* represent the intra-processor part boundaries

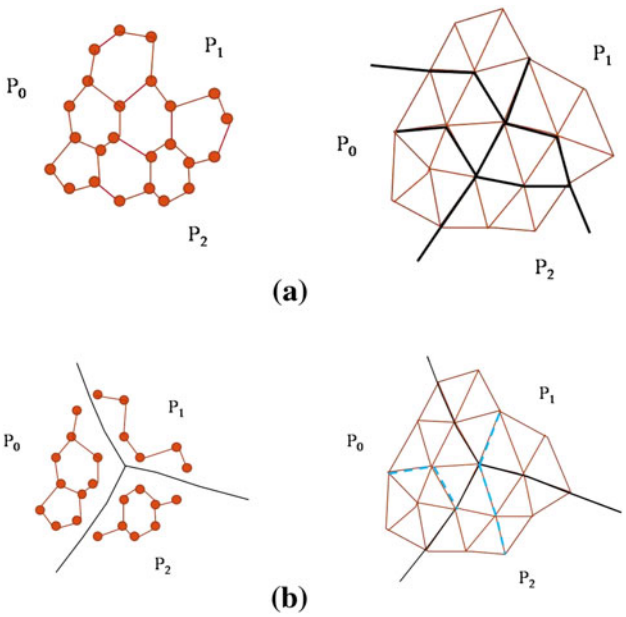

Fig. 5 The graph (left) and the partition (right) given by **a** global partitioning and **b** local partitioning

partitioning based on original starting partition in Fig. 5b. In the global partitioning, the redistribution is based on the global graph information (left of Fig. 5a). The original three part inter-processor part boundary is not maintained in the redistribution for global partitioning (i.e., mesh entities are migrated between parts on same and different processors). Conversely, in the local partitioning, the redistribution is based on the intra-edges of the graph without knowing the existence of the other processors, and the inter-processor part boundaries stay fixed (i.e., there is no migration of mesh entities between processors).

Local partitioning requires much smaller compute times when compared to global partitioning, especially for a partition with a large number of graph nodes or a large number of parts, where a global partitioning implementation may even fail. However, as local partitioning is repeated, the quality of the partition will be decreased due to the compounding effects, since the balance of a partition uses only limited load information, thus limiting the quality of the partition relative to global partitioning. Local partitioning is not optimal in terms of balance of the mesh entities that define the graph nodes, but provides good partitions that can easily be improved by iterative algorithms which are also capable of accounting for multiple criteria [6]. Two example uses of multiple parts per processor with different combinations of global and local partitioning demonstrate common usage of multiple parts per processor.

3.1 AAA model with O(10^8) elements

In the first example a mesh with 133 million elements on a patient-specific abdominal aortic aneurysm (AAA) model is considered (as shown in Fig. 6). The mesh is obtained through adaptive refinement of an anisotropic mesh obtained from previous adaptation cycles. In this case, scaling studies are performed on partitions containing from 1,024 parts to 65,536 parts, and thus, using 1,024 to 65,536 processing cores, respectively, on Intrepid (IBM Blue Gene/P) at ALCF, ANL. To create all the mesh partitions, a global partitioning strategy is applied and we use the graph partitioner, ParMetis [20, 21]. The partitioning starts with a well-balanced mesh (2.7% region imbalance) distributed on 512 parts. The repartitioning is carried out on 512 opteron processors Kraken-XT4 (Cray XT4) at NICS. Table 1 lists the average number of regions and vertices of each partition along with imbalance ratio.

Due to the "global" partitioning, the imbalance ratio of the regions (the part-objects) satisfies the imbalance tolerance globally (the 4th column of Table 1), which is set at 1.03. After repartitioning, each processor contains multiple parts (e.g., 16 parts per processor in the 8,192 part partition).

The scaling study of the finite element solver, PHASTA, is performed on Intrepid using all the 4 cores on each compute node, and allowing one part on each core. A Womersley profile [22] is applied at the inlet and rigid vessel walls with no-slip boundary conditions are considered. At outlets a lumped parameter boundary condition [23] is applied. The study involves 20 time steps and the time usage for different partitions is measured and the scaling factor is computed (i.e., s-factor $= t_{\text{base}} \times np_{\text{base}}/(t_i \times np_i)$) (see Table 2). A near-perfect strong scaling factor of the finite element solver is observed as we increase the number of processing cores from 1,024 up to

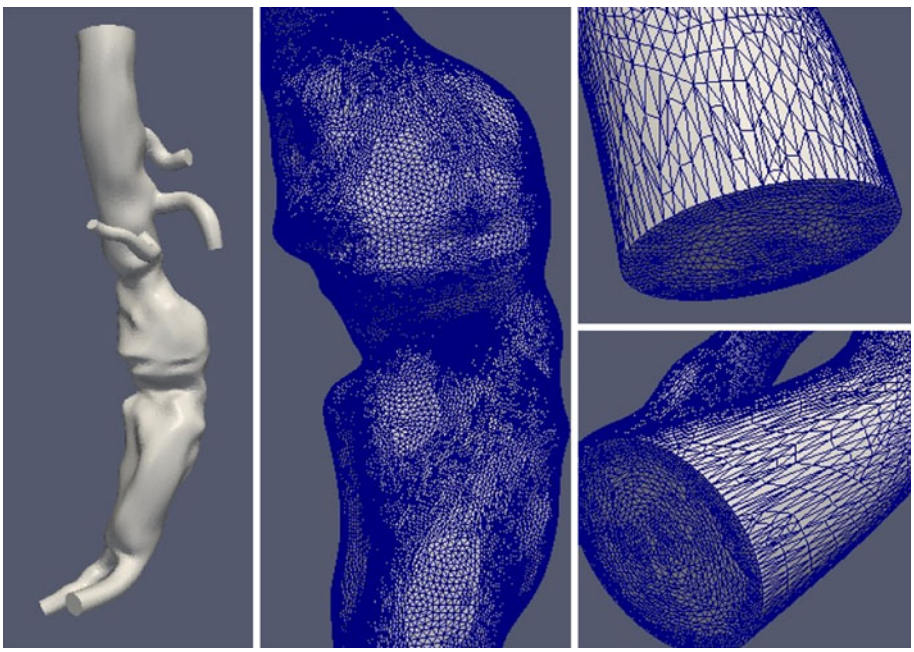

Fig. 6 Geometry and a mesh of an AAA model

Table 1 The average number of regions and vertices of a 133 million mesh of an AAA model, along with the imbalance ratios

Num of parts	Avg. num. rgns	Avg. num. vtxs	Rgn. imbal.	Vtx. imbal.
1,024	130,833	24,413	1.023	1.053
2,048	65,417	12,555	1.023	1.061
4,096	32,708	6,498	1.019	1.088
8,192	16,354	3,392	1.019	1.098
16,384	8,177	1,781	1.020	1.130
32,768	4,088	948	1.023	1.158
65,536	2,044	509	1.024	1.203

32,768, see the last column in Table 2. The time usage of the two main components of the finite element analysis, equation formation and equation solution, is measured and listed in the table. A near-perfect scaling is observed for system formation all the way up to 65,536 parts, while a degradation of equation solution is observed above 32,768 processing cores. This is due to the increased vertex imbalance for a fixed-size problem distributed on a large number of parts. A 20.3% vertex imbalance is observed on the 65,536 part partition, however, the number of spikes (heavily loaded parts) with more than 10% vertex imbalance is only 481 (481/65,536 = 0.7%). Algorithms based on iterative load balancing operations between neighboring parts have been developed in [6] to eliminate this small number of spikes to improve the partition and in turn parallel performance.

3.2 AAA model with O(10^9) elements

As a second case, we consider a relatively large mesh with O(10^9) elements for an AAA model (as shown in Fig. 6). Again, the mesh is obtained through adaptive refinement of an anisotropic mesh obtained from previous adaptation cycles. In this case, scaling studies are also performed on the near-petascale system, Intrepid (IBM BG/P system) at ALCF, ANL. The number of cores covered range from 4,096 to 163,840, i.e., from 1 rack to all 40 racks of BG/P covering the full system-scale. To be able to create all the mesh partitions, we used the Zoltan HyperGraph partitioner PHG [13], and a local partitioning strategy is applied. The average number of mesh regions and vertices for each partition along with imbalance ratios are given in Table 3. The local partitioning starts with a well-balanced (2.49% region imbalance) mesh on 2,048 parts, which is obtained by globally partitioning using ParMetis. Each part of the 2,048 parts is split to multiple parts, for example, from 1 to 80 parts on each processor when the mesh is redistributed from 2,048 to 163,840 parts. The local partitioning is more efficient, especially for the case with large number of mesh entities and large number of parts, however, the compounding effect causes the imbalance ratio to increase. We see this compounding effect easily from the fourth column in Table 3. A 1.03 imbalance tolerance is set up in the local partitioning and an imbalance ratio of 1.0549 is obtained in the final partitions. The local partition is performed on 2,048 opteron processing cores on Kraken (Cray XT5) at NICS. The vertex imbalance of the 163,840 part partition goes up to 35.71%, which is detrimental to the scaling of equation solution phase of the finite element analysis tool. If we take a partition with 4,096 parts which is obtained by the global partitioning procedures and then apply local partitioning on it (each of the 4,096 part split to 40 parts), a slightly better partition is obtained (from the standpoint of vertex imbalance). The vertex imbalance is 19.5% with 1,561 as average number of vertices per part and the region imbalance is 5.54%, which is expected due to the compounding effect. This partitioning (4k to 160k) is performed on 4,096 Kraken processing cores. The information of this partition is listed in the last row of Table 3 and it is the one used in the scaling study.

The same problem set up as before is used and 20 time steps are solved in the scaling study. Here, we measure the execution time in terms of equation formation and equation solution so as to understand the effect of partitioning on scaling, specifically due to the imbalance in mesh vertices. Figure 7 and Table 4 provide the scaling factor and execution time (including for both work components). We

Table 2 Strong scaling results on an AAA model with 133M elements on Intrepid:IBM BG/P (execution time is measured in seconds)

133M elements		Eqn. form		Eqn. soln		Total	
Num. of core	Avg. elem./core	Time	*s*-factor	Time	*s*-factor	Time	*s*-factor
1,024(base)	130,833	191.98	1.0	253.72	1.0	445.7	1.0
2,048	65,417	95.69	1.003	127.35	0.996	223.0	0.999
4,096	32,708	47.67	1.007	63.73	0.995	118.2	1.000
8,192	15,354	23.89	1.004	33.15	0.957	57.04	0.977
16,384	8,177	11.90	1.008	16.89	0.939	28.80	0.967
32,768	4,088	6.02	0.996	8.87	0.894	14.89	0.935
65,536	2,044	3.04	0.987	5.26	0.754	8.30	0.839

Table 3 The average number of regions and vertices of a 1.07 billion element mesh of an AAA model, along with the imbalance ratios

Num of parts	Avg. num. rgns	Avg. num. vtxs	Rgn. imbal.	Vtx. imbal.
4,096 (base)	261,667	48,054	1.0549	1.0736
8,192	130,834	24,732	1.0549	1.0841
16,384	65,417	12,822	1.0549	1.0990
32,768	32,708	6,707	1.0549	1.1210
65,536	16,354	3,550	1.0549	1.1719
98,304	10,903	2,465	1.0549	1.2117
131,072	8,177	1,908	1.0549	1.1782
163,840	6,542	1,570	1.0549	1.3571
4k to 160k	6,542	1,561	1.0554	1.1950

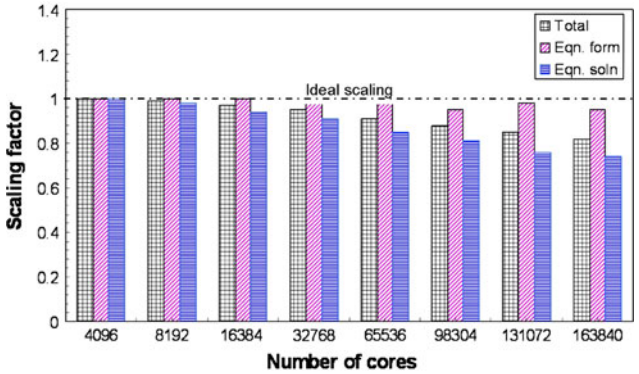

Fig. 7 Strong scaling performance for an AAA model with $O(10^9)$ elements up to 163,840 cores of Intrepid, IBM BG/P system. In this case, execution time is measured in terms of equation formation, equation solution and the total

observe a near-perfect strong scaling (95% or above) for equation-formation stage all the way to the full system-scale. On the other hand, strong scaling for equation-solution stage drops below 90% at 65,536 cores and onwards. The strong scaling for both work components together is at 90% or above until 65,536 cores and drops to 82% while using 163,840 cores or all the 40 racks of BG/P in Intrepid. Therefore, with 20 time steps in this case, each step takes only 1.25 s for an implicit solve (over 163,840 cores) on an adapted mesh with 1B elements (where, B denotes billion).

It is important to point out that the imbalance in mesh vertices goes up from 7% in the partition with 4,096 parts to 17% in the case of 65,536 parts; and is as high as 19.5% in the partition with 163,840 parts. Furthermore, note that the average number of mesh vertices per part (or core) relatively increases when a fixed-size problem is spread into more and more parts, for example, 48,000 is the average number of mesh vertices per part in the partition with 4,096 parts and is 1,561 on average in the case of 163,840 parts; which is about 30% higher relatively (1,200 mesh vertices will be ideal based on 48,000 when going from 4,096 parts to 163,840 parts). This relative growth in the average number of mesh vertices per part on higher partitions leads to a situation where computational work is not properly scaled as compared to the base.

The same strong scaling study has been performed on Kraken at NICS using up to 98,304 processor cores. The results are demonstrated in Table 5. Consistent with the observation on Intrepid, the system formation scales all the way up to 98,304 processing cores, however, a degradation of equation solution is observed much earlier due to the architecture differences.

By adding the iterative load balancing procedure given in [6] to eliminate this small number of spikes in vertex imbalance, the scaling and parallel performance of locally balanced meshes can be enhanced to closely match that obtained when global balancing plus iterative load balancing are used.

4 Predictive load balancing

The goal of mesh adaptation is to modify the mesh so that the element sizes and distribution provide the desired levels of accuracy for nearly an optimum number of elements. To achieve this goal the manner in which the mesh is refined or coarsened varies over the domain. Thus a mesh that is well balanced before mesh adaptation will typically be poorly balanced after mesh adaptation. Thus the mesh must be repartitioned before the next analysis step. It was found that doing this repartitioning after parallel mesh adaptation caused problems since some parts would have most of the mesh refinement and they will often exceed the limit of the physical memory of these processors. This either slows down or kills the process. To avoid this problem a predictive load balancing step, applied before mesh adaptation, can be introduced to redistribute the mesh such that it will be nearly balanced after mesh adaptation. In this predictive load balancing step a weight is specified to each of the current elements based on a prediction of how many elements will fill the space of that element. In [1, 24] these weights were determined based on the 3D subdivision patterns after the edges have been marked for splitting. A similar technique has also been used by Oliker et al. [25].

The predictive load balance (*PredLB*) algorithm introduced herein uses an a priori estimate of the number of mesh regions after mesh adaptation given a general anisotropic mesh size field and accounts for both mesh refinement and coarsening. The algorithm has been added to mesh modification with the goal of achieving about equal number of regions in the adapted mesh on each part.

Table 4 Strong scaling results on an AAA model with O(10^9) elements up to 163,840 cores on Intrepid:IBM BG/P (execution time is measured in seconds)

Intrepid: 1.07B element		Eqn. form.		Eqn. soln.		Total	
Num. of core	Avg. elem./core	Time	*s*-factor	Time	*s*-factor	Time	*s*-factor
4,096 (base)	261,600	388.68	1	456.24	1	844.92	1
8,192	103,800	194.48	1.00	233.86	0.98	428.34	0.99
16,384	65,400	97.48	1.00	120.90	0.94	218.38	0.97
32,768	32,700	49.01	0.99	62.33	0.91	111.34	0.95
65,536	16,350	24.59	0.99	33.70	0.85	58.29	0.91
98,304	10,900	16.42	0.99	23.56	0.81	39.98	0.88
131,072	8,175	12.37	0.98	18.65	0.76	31.02	0.85
163,840	6,540	10.27	0.95	15.34	0.74	25.61	0.82

Table 5 Strong scaling results on an AAA model with O(10^9) elements up to 65,536 cores on Kraken: Cray XT5

Kraken: 1.07B element		Eqn. form.		Eqn. soln.		Total	
Num. of core	Avg. elem./core	Time	*s*-factor	Time	*s*-factor	Time	*s*-factor
4,096 (base)	261,600	63.57	1	247.77	1	311.34	1
8,192	103,800	31.98	0.996	112.25	1.104	144.23	1.079
16,384	65,400	15.50	1.025	57.56	1.076	73.06	1.065
32,768	32,700	7.86	1.010	32.10	0.965	39.35	0.974
65,536	16,350	4.01	0.991	24.03	0.644	28.04	0.694
98,304	10,900	2.69	0.985	15.98	0.646	18.67	0.695

Figure 8 demonstrates the number of mesh regions after mesh modification with and without using *PredLB*.

4.1 Predictive load balancing algorithm

Anisotropic mesh adaptation [26, 27] requires a way to define the desired element size distribution over the domain. Mesh metric tensors are used to define a desired anisotropic mesh size field in the anisotropic mesh adaptation [28–30]. The concept of a mesh metric field is used to represent the desired size field as a tensor over the domain. The mesh metric tensor at any point P in the domain is defined as a symmetric positive definite matrix $\mathbf{T}(P)$. The associated quadratic form $\langle \mathbf{x}, \mathbf{Tx} \rangle = 1$ defines a mapping of an ellipsoid in the physical space into a unit sphere in the transformed/metric space (Fig. 9). In other words, any vector x at point P assumes a unit value in the space where any distance is measured under the metric $\mathbf{T}(P)$.

The ideal goal of the mesh adaptation is a mesh with regular elements in the metric space where each edge e must satisfy the following equality:

$$\langle e, \mathbf{T}e \rangle = 1 \tag{1}$$

A mesh with all its edges satisfying this relationship is referred as a *unit mesh*. However, the fact that we cannot

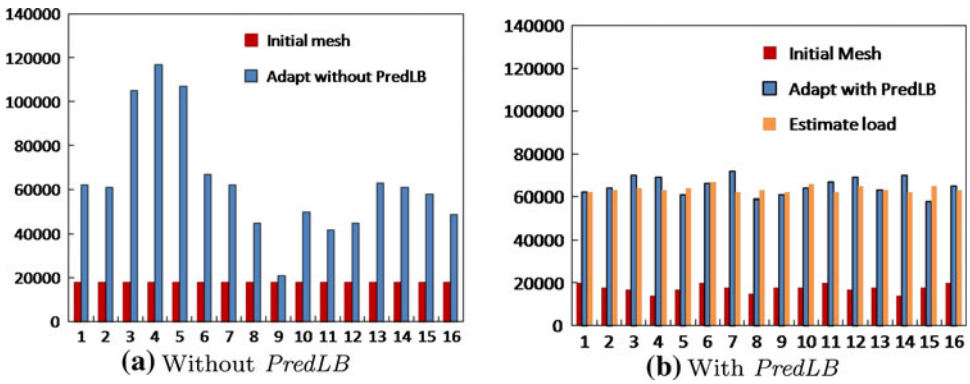

Fig. 8 A demonstration of the predictive load balance algorithm

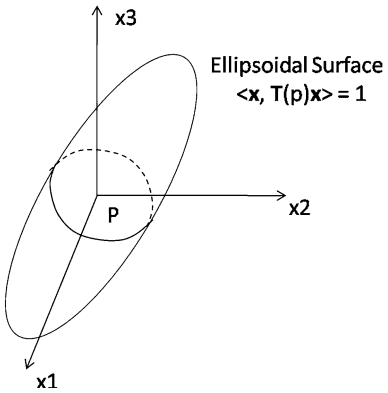

Fig. 9 Ellipsoidal surface associated with the quadratic form

pack unit regular tetrahedral to satisfy constant unit mesh metric field over a domain indicates that a perfect matching is not possible. Therefore, a relaxed criteria is inevitable [26, 31]. During mesh adaptation the mesh metric tensor $\mathbf{T}(P)$ is decomposed [31] into three unit directions (e_1, e_2, e_3) and the desired length h_1, h_2, and h_3 in each corresponding direction. The desired element volume (tetrahedral element) at point P is then computed as $V = h_1h_2h_3/6$.

To accomplish the goal of *PredLB*, weights have to be specified to graph nodes, which correspond to the elements before refinement, and repartitioning on a weighted graph performed. The weight is set based on an estimate of the number of elements to be generated in the volume of that element.

Pain et al. [32] use the ratio of the element volume of the mesh before adaptation to the volume of desired element in the adapted mesh to estimate the number of elements to be generated in the mesh refinement. A similar idea combined with the concept of the mesh metric field is used to specify the weight of the graph nodes for *PredLB*. Given the initial mesh and mesh metric field attached to the vertex the desired element volume at each vertex P is computed as $h_1(P)h_2(P)h_3(P)/6$. The average volume of the desired elements within the current element volume is estimated as $\sum_{p=1}^{n_{en}} h_1(p)h_2(p)h_3(p)/6n_{en}$, where n_{en} is the number of vertices per element, which is 4 for tetrahedral mesh. The number of adapted elements within each of the current element is estimated by the volume of the current mesh element over the average volume of the desired mesh elements expressed as,

$$num = \frac{R\_volume}{\sum_{p=1}^{n_{en}} h_1(p)h_2(p)h_3(p)/6n_{en}} \tag{2}$$

These numbers are specified as weights to the current elements which are defined as the graph nodes.

Some graph-based partitioners take only single precision integers as the object weight, which is constraining. To avoid rounding of fractional weights (including to zero), the smallest predicted weight is normalized to 1, which can make either the maximum weight or the summation on weights internal to the partitioner easily overflow a 4 byte integer. An artificial clip on either side of the "integer converter" is required to resolve this problem. Equations 3, 4 and 5 are the modifications to *num* needed to scale them into a good range. Instead of the smallest weight, a larger value is used to normalize the predictive weight to keep the average from growing too high, and all those ones whose weights are less than one are promoted up to one to avoid zeros.

The following formulas are used to rescale *num* so as to guarantee the weight specified to the graph nodes to be in a good range of (4 byte) integers. *if* ($max(num)/min(num) > maxvwgt$)

$$nvwgt = \frac{num}{max(num)/maxvwgt}, \tag{3}$$

else

$$nvwgt = \frac{num}{min(num)}, \tag{4}$$

If ($nvwgt < 1$)

$$nvwgt = nvwgt + 1, \tag{5}$$

where *maxvwgt* is the prescribed maximum weight, $min(num)$ and $max(num)$ are the minimum and the maximum value of *num*, respectively, over the partition.

4.2 Applications of predictive load balance

Examples of mesh adaptation on two geometric models are considered to study the performance of predictive load balance. The first one is the AAA model (shown in Fig. 6) and the second test is on a straight pipe model with air bubbles distributed in the pipe. The first test is performed on an Opteron cluster, and the second is performed on an IBM Blue Gene/L at Rensselaer's CCNI.

In the first example, an anisotropic adaptation on an AAA model is considered. The initial mesh with 8.7 million regions in total is distributed on 128 parts. The solution-based anisotropic adaptation is carried out on the initial mesh to increase the local mesh resolution in order to capture the flow features better. A mesh with 80.5 million regions in total is obtained and the adaptation is performed on 128 AMD Opteron processor cores. Figure 10a demonstrates the number of regions on each part before and after mesh adaptation and compares that with and without the use of *PredLB*. Without *PredLB*, we see peaks (high number of regions) in the region distribution which uses more memory than other processors. Using *PredLB*, the number of regions on each part becomes more even

(with lesser variation). However, on part 18, the number of regions is a little higher than others. This is due to an underestimate of the adapted number of regions by *PredLB* algorithm in the coarsening process. Figure 10b shows the memory usage (the high water mark) before and after mesh adaptation with and without using the *PredLB* algorithm. The memory usage after adaptation is much higher without *PredLB*.

Anisotropic mesh adaptation with a new mesh size field is performed again on the 8.7 million region mesh of the AAA model which, this time, is distributed on 200 AMD Opteron cores. The adapted mesh in this case has 187 million regions. Figure 11 shows the number of regions on each part before and after mesh adaptation. Without *PredLB* the test runs out of memory making it impossible to compare.

The second example is on a straight pipe model with air bubbles distributing in the pipe, which is a physically relevant model (e.g., 2 phase flow simulation). The isotropic mesh size field which "tracks" the "air bubbles" in the model is shown in Fig. 12. The blue color demonstrates small values of size field which implies a fine mesh (or high resolution) and the red color demonstrates relative large size field which implies a coarse mesh. In multi-phase flow simulations, meshes with high resolution at phase boundaries are desired to capture complex physical phenomena at the interface.

The description of the size field of an air bubble is as follow:

for $(0 < R < R_0)$

$$h_0 = h_1 = h_2 = c_1(1.0 - e^{-abs(R-R_0)L}) + c_2, \quad (6)$$

else

$$h_0 = h_1 = h_2 = c_1 + c_2, \quad (7)$$

where R_0 is the radius of the air bubble and R is the distance of a point to the center of the nearest air bubble, c_1, c_2 and L are constants. The selected definition $(c_1 + c_2)$ of the mesh size field outside the air bubble regions gives a smooth transition in the mesh size.

A segment of the straight pipe model with air bubbles is shown in Fig. 13a. The mesh adaptation starts with an initial uniform tetrahedral element mesh and obtains non-uniform mesh with air bubbles. Figure 13b and c are examples of initial mesh and adapted mesh zoomed for a bubble colored by the magnitude of size field. To demonstrate the usefulness of predictive load balancing, the first test on this straight pipe model compares the number of regions on each part in the adapted mesh with and without using *PredLB*. The initial mesh is well balanced on 1,024 parts with 1.24 million as the total number of mesh regions which is adapted to a 36.6 million region mesh with five air bubbles distributed in the straight pipe. The number of regions before and after the mesh adaptation with and without using *PredLB* is demonstrated in Fig. 14. Without *PredLB*, the number of regions on some parts are much

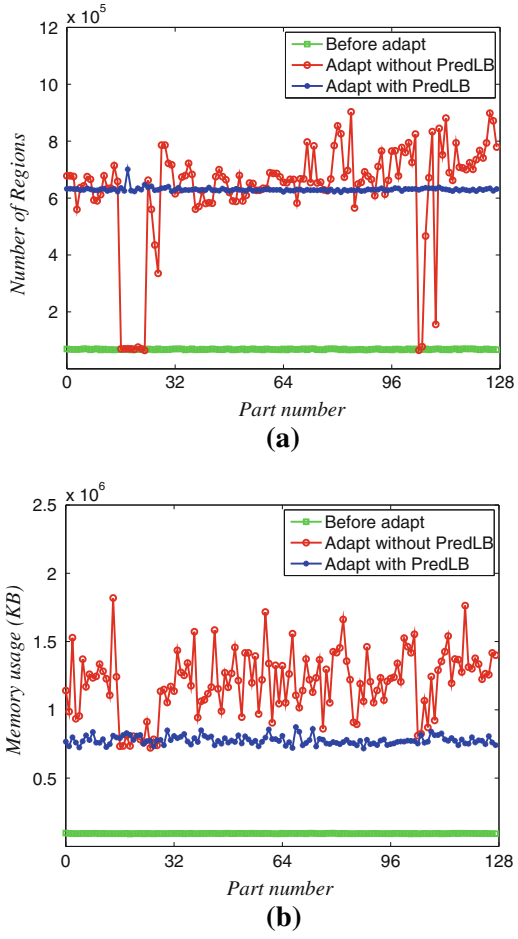

Fig. 10 **a** Number of regions and **b** memory usage (high water mark) on each part before and after mesh adaptation with and without using *PredLB* on an AAA model

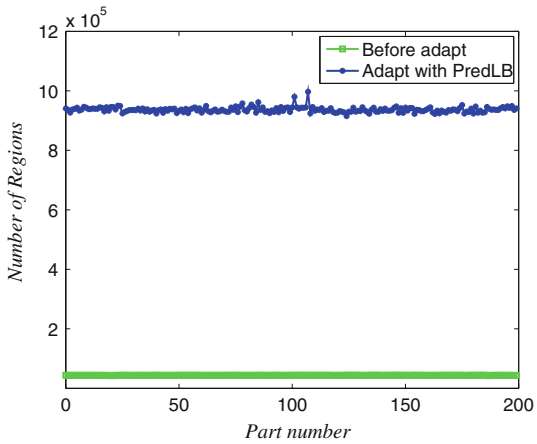

Fig. 11 The number of regions on each part before and after mesh adaptation with predictive load balance on an AAA model

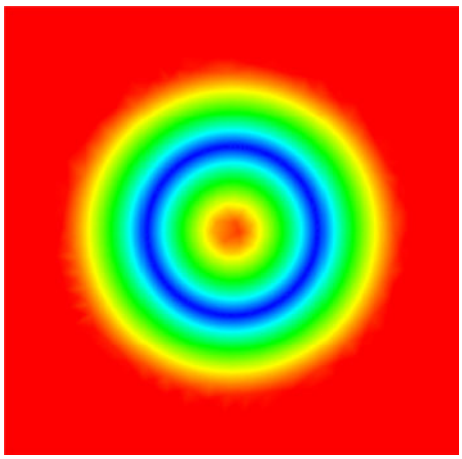

Fig. 12 The size field of an "air bubble"

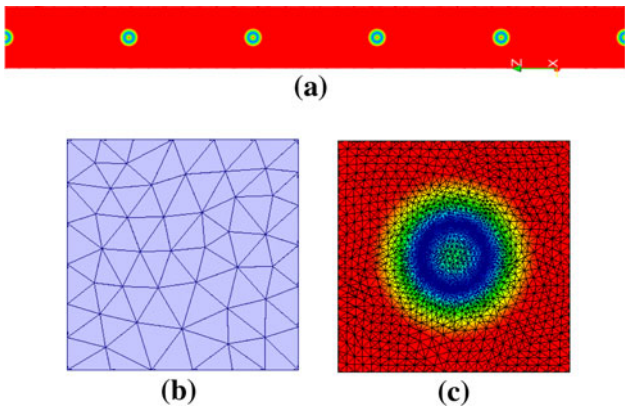

Fig. 13 A segment of a straight pipe model with distributed air bubbles (**a**) colored by the magnitude of size field along with examples of initial mesh (**b**) and adapted mesh zoomed for one bubble (**c**)

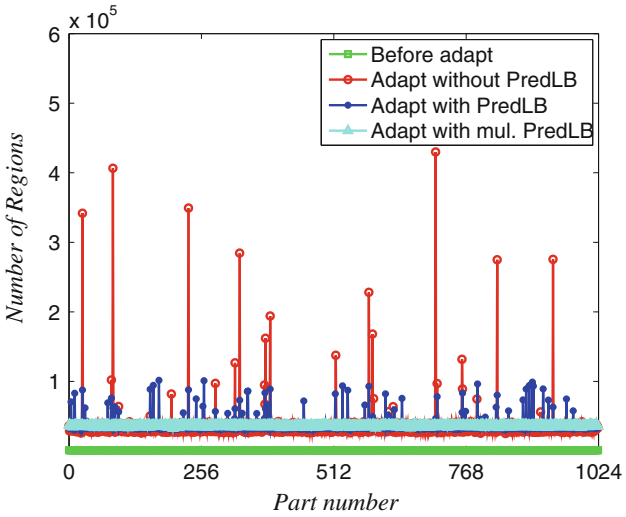

Fig. 14 The number of regions on each part of the 1,024 parts before and after mesh adaptation with and without *PredLB* on a straight pipe model with five air bubbles

higher than others (red line) after mesh adaptation. The balance of mesh regions in the adapted mesh is improved by using *PredLB* (blue line), however, it is still not optimal. Unlike the AAA model case, an analytical size field is utilized in this test, which is updated (recomputed using (6) and (7)) after each iteration for newly created mesh entities. This updating makes it more difficult to accurately estimate the number of mesh regions in the final mesh due to a very coarse initial mesh, which is the situation in this study. If the mesh adaptation is performed on supercomputers with relatively smaller memory per core (e.g., BG/L), then the *PredLB* can be used multiple times (i.e., used each time in the first several iterations) to further improve the balance in the final mesh (cyan line in Fig. 14), hence avoiding running out of memory in the whole adaptation process.

In a much bigger test, a uniform initial mesh with 17.2 million regions is well balanced on 16,384 parts and adapted to 1.06 billion regions using 16,384 IBM Blue Gene/L cores with multiple predictive load balance. 160 air bubbles are distributed in the pipe in the adapted mesh. Figure 15 shows the number of regions on each part before and after adaptation with multiple *PredLB* steps. The process runs out of memory without *PredLB*.

5 Closing remarks

This paper has presented two capabilities that were developed to support the ability to perform parallel adaptive simulations on massively parallel computers.

The multiple parts per processor tool effectively supports a number of possible operations including effectively changing, in parallel, the number of processors being used during a simulation. As the size of the adaptive mesh grows one can select points in the simulation to create multiple parts per processor, and then distribute the additional parts to new processing cores. Of course, one could reduce the number of processing cores being used by sending parts to selected cores where they are merged back into one part and the emptied processing cores released. To be effective, algorithms would need to be added to merge parts that when combined would represent a reasonably good part in terms of required inter-processor communications. The partition model could be effectively used in the development of such an algorithm.

The predictive load balancing tool has become an integral part of the mesh modification procedure used for the construction of the adapted meshes. Without it many of the adaptive simulations simply failed due to hitting memory limits.

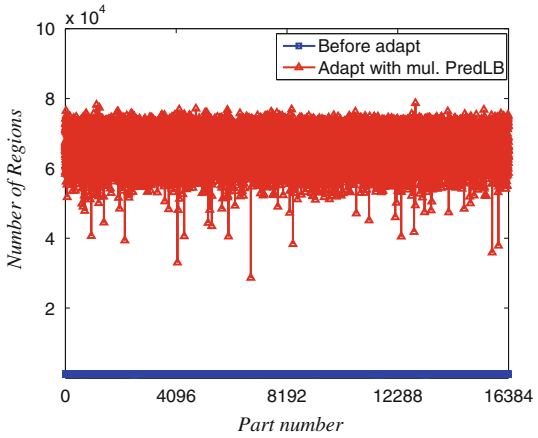

Fig. 15 The number of regions on each part of the 16,384 parts before and after mesh adaptation with and without *PredLB* on a straight pipe model with 160 air bubbles

Note that in future investigations it may be useful to consider the combination of predictive load balancing with the multiple parts per processor by increasing the number of parts based on the mesh before partitioning with consideration of the weights used in predictive load balancing. To have a chance of being effective, the number of parts created per processor would need to vary between processing cores.

Acknowledgments We gratefully acknowledge the support of this work by NSF's PetaApps program under grant OCI-0749152 and by the DOE Office of Science's SciDAC/ITAPS program under grant DE-FC02-06ER25769. This research used computing resources provided by: (a) Rensselaer's Computational Center for Nanotechnology Innovations that is funded by the State of New York, IBM and Rensselaer Polytechnic Institute, (b) Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the DOE under contracts DE-AC02-06CH11357, and (c) NSF through TeraGrid resources provided by National Institute for Computational Sciences (NICS); Kraken system was used. We would like to acknowledge that results presented in this article made use of software components provided by ACUSIM Software Inc. and Simmetrix Inc.

References

1. Flaherty J, Loy R, Özturan C, Shephard M, Szymanski B, Teresco J, Ziantz L (1996) Parallel structure and dynamic load balancing for adaptive finite element computation. Appl Numer Math 26:241–263
2. Shephard M, Flaherty J, Bottasso C, de Cougny H, Ozturan C, Simone M (1997) Parallel automated adaptive analysis. Parallel Comput 23:1327–1347
3. Sahni O, Zhou M, Shephard M, Jansen K (2009) Scalable implicit finite element solver for massively parallel processing with demonstration to 160k cores. In: Proceedings of the SC09. Springer, Berlin
4. Shephard M, Jansen K, Sahni O, Diachin L (2007) Parallel adaptive simulations on unstructured meshes. J Phys Conf Ser 78:012053
5. Zhou M, Sahni O, Kim H, Figueroa C, Taylor C, Shephard M, Jansen (2010) Cardiovascular flow simulation at extreme scale. Comput Mech 46:71–82 doi:10.1007/s00466-009-0450-z
6. Zhou M, Sahni O, Shephard M, Devine K, Jansen K (2010) Controlling unstructured mesh partitions for massively parallel simulations. SIAM J Sci Comput 32(6):3201–3227
7. Beall MW, Shephard MS (1997) A general topology-based mesh data structure. Int J Numer Methods Eng 40:1573–1596
8. Celes ERW, Paulino GH (2005) A compact adjacency-based topological data structure for finite element mesh representation. Int J Numer Methods Eng 64:1529–1565
9. Garimella R (2002), Mesh data structure selection for mesh generation and fea applications. Int J Numer Methods Eng 55:451–478
10. Seol E, Shephard M (2006) Efficient distributed mesh data structure for parallel automated adaptive analysis. Eng Comput 22(3–4):197–213
11. Chand K, Diachin L, Li X, Ollivier-Gooch C, Seol E, Shephard M, Tautges T, Trease H (2008) Toward interoperable mesh, geometry and field components for PDE simulation development. Eng Comput 24(2):165–182
12. Devine K, Diachin L, Kraftcheck J, Jansen KE, Leung V, Luo X, Miller M, Ollivier-Gooch C, Ovcharenko A, Sahni O, Shephard M, Tautges T, Xie T, Zhou M (2009) Interoperable mesh components for large-scale, distributed-memory simulations. J Phys Conf Ser 180:012011
13. Boman E, Devine K, Fisk L, Heaphy R, Hendrickson B, Leung V, Vaughan C, Catalyurek U, Bozdag D, Mitchell W (1999) Zoltan home page. http://www.cs.sandia.gov/Zoltan
14. Devine K, Boman E, Heaphy R, Hendrickson B, Vaughan C (2002) Zoltan data management services for parallel dynamic applications. Comput Sci Eng 4(2):90–97
15. Devine K, Boman E, Heaphy R, Bisseling R, Catalyurek U (2006) Parallel hypergraph partitioning for scientific computing. In: Proceedings of 20th international parallel and distributed processing symposium (IPDPS06), IEEE
16. Karypis G, Kumar V (1999) Parallel multilevel k-way partitioning scheme for irregular graphs. SIAM Rev 41(2):278–300
17. Pellegrini F (2007) Scotch 5.0 user's guide. Tech. rep., LaBRI
18. Walshaw C, Cross M (2007) JOSTLE: parallel multilevel graph-partitioning software—an overview. In: Mesh partitioning techniques and domain decomposition techniques, pp 27–58
19. Teresco J, Devine K, Flaherty J (2005) Partitioning and dynamic load balancing for the numerical solution of partial differential equations. In: Numerical solution of partial differential equations on parallel computers. Springer, Berlin
20. Karypis G, Kumar V (1996) A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. In: 10th intlernational parallel processing symposium, pp 314–319
21. Karypis G, Kumar V (1998) Multilevel algorithms for multi-constraint graph partitioning. In: Proceedings of the 1998 ACM/IEEE conference on supercomputing, pp 1–13
22. Womersley J (1955) Method for the calculation of velocity, rate of flow and viscous drag in arteries when the pressure gradient is known. J Physiol 127:553–563
23. Vignon-Clementel I, Figueroa C, Jansen K, Taylor C (2006) Outflow boundary conditions for three-dimensional finite element modeling of blood flow and pressure in arteries. Comput Methods Appl Mech Eng 195(29–32):3776–3796
24. Flaherty J, Loy R, Shephard M, Szymanski B, Teresco J, Ziantz L (1997) Predictive load balancing for parallel adaptive finite element computation. In: Proceedings of the international conference on parallel and distributed processing techniques and applications, PDPTA'97, vol 1, pp 460–469
25. Oliker L, Biswas R, Strawn R (1996) Parallel implementation of an adaptive scheme for 3D unstructured grids on the SP2. In:

Proceedings of the 3rd international workshop on parallel algorithm for irregularly structured problems
26. Li X, Shephard M, Beall M (2005) 3D anisotropic mesh adaptation by mesh modification. Comput Methods Appl Mech Eng 194:4915–4950
27. Sahni O, Müller Y, Jansen K, Shephard M, Taylor C (2006) Efficient anisotropic adaptive discretization of the cardiovascular system. Comput Methods Appl Mech Eng 195:5634–5655
28. Borouchaki H, George P, Hecht F, Laug P, Saltel E (1997) Delaunay mesh generation governed by metric specifications. part i. algorithms. Finite Elem Anal Des 25(1–2):61–83
29. Frey P, George PL, Hecht F (2008) Mesh generation: application to finite elements, 2nd edn. ISTE, London
30. Labbe P, Dompierre J, Vallet M-G, Guibault F, Trpanier J-Y (2004) A universal measure of the conformity of a mesh with respect to an anisotropic metric field. Int J Numer Methods Eng 61(16):2675–2695
31. Li X (2003) Mesh modification procedures for general 3D non-manifold domains. Ph.D. thesis, Rensselaer Polytechnic Institute
32. Pain C, Umpleby A, de Oliveira C, Goddard A (2001) Tetrahedron mesh optimization and adaptivity for steady-state and transient finite element calculations. Comput Methods Appl Mech Eng 190:3771–3796