Parallel mesh control tools for unstructured meshes

Min Zhou,1 Aleksandr Ovcharenko,1 Ting Xie,1 Seegyoung Seol,1 Kenneth E. Jansen,2 Onkar Sahni,1 and Mark S. Shephard1*

1Rensselaer Polytechnic Inst, Sci Computat Ctr, Troy, NY12180
2University of Colorado at Boulder, Boulder, CO80309

E-mail: *shephard@scorec.rpi.edu

Abstract. The ITAPS center is developing services for unstructured mesh operations that operate on massively parallel computers including base services to effectively support evolving mesh partitions that take advantage of local and/or neighborhood information to eliminate synchronization steps and reduce communications. This paper overviews base services for controlling mesh partitions and controlling messages that are communicated within processor neighborhoods.

1. Introduction

Unstructured adaptive mesh methods support the effective and reliable analysis of complex physical behaviors modeled by partial differential equations over general three-dimensional domains. 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 cores of the parallel computer through a partitioning of the mesh in which the amount of computation required for each part is nearly equal and the amount of inter-processor communication between parts is as small as possible. Given the fact that the meshes needed will not fit within the memory of even a single node, the methods developed to manage the mesh in parallel must address: (i) Understanding how the mesh entities and their adjacencies relate to computation and/or memory use. (ii) Performing multiple dynamic load balancing steps to account for the changing load balance caused by operations as adaptive mesh modification. (iii) Constructing and managing the mesh partition must be performed using effective scalable methods to avoid that step becoming more dominant as we move toward exascale machines.

A key conclusion the above factors point to 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 [3,4,5] 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 when the total size of the mesh grows by one or two orders of magnitude during a
simulation, yielding a situation where it is desirable to increase the number of cores being used in subsequent 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.

2. Partition Control on Massively Parallel Computers

Graph/hypergraph-based algorithms are most effective for partitioning unstructured meshes [7,8,9]. In our work on parallel computations at extreme scale, specific issues were observed [6]. One issue relates to the cost and reliability of applying parallel dynamic load balance over all cores using a graph of the entire distributed mesh, which is referred to as global partitioning. As core counts increase, the quality of global partitions can degrade. In addition there is a reliability issue that arises in some current graph/hypergraph-based partitioner implementations that, in our experience, often fail for partitions requiring large number of parts (≥100K). The second issue is related to the characteristics of implicit parallel mesh-based analysis which has the two primary work components of equation formation and equation solution which require load balance of different types of mesh entities (elements and vertices in the present case). Graph/hypergraph-based partitioning uses one type of mesh entities as graph nodes, thus, the balance of other mesh entities may not be optimal. Partition improvement methods have been developed to account for a balance of multiple entity types within a single partition. Those issues are addressed in our work by the combined strategy of global and local graph partitioning plus iterative improvement using the algorithm developed in [6].

Two partitioning categories are defined based on data provided to graph/hypergraph-based partitioner: global partitioning which considers both intra-processor and inter-processor graph edges, and local partitioning which considers only intra-processor graph edges. Global partitioning considers the complete set of graph edges and provides a balanced partition with well controlled inter-part communication. Local partitioning considers only the on-processor (intra-processor) graph edges and nodes, without knowing the existence of graph nodes and edges on other processors. In this case, partitioning is carried out independently on each processor, as a serial process, which can be executed in an embarrassingly parallel fashion on all processors. At large core counts local partitioning requires much smaller compute time where a global partitioning implementation may fail. However, as local partitioning is repeated, the quality of the partition will decrease due to the compounding of imbalance in each step. Local partitioning is not optimal, but provides good starting partitions that can easily be improved by iterative algorithms.

When very large core counts are considered, the problems observed in the partition obtained from the graph/hypergraph-based procedures are limited to a number of heavily loaded parts (based on mesh vertices), referred to as spikes. Thus, scalability of the equation solution phase is limited by these spikes. The diffusive approach, Local Iterative Inter-Part Boundary Modification (LIIPBMod) developed in [6], reduces spikes by migrating selected mesh entities from relatively heavily loaded parts to less loaded neighboring parts. On heavily loaded parts, the mesh vertices on the inter-part boundary are traversed and the ones bounding a small number of elements are identified. Then the elements adjacent to selected vertices are migrated to the lightly loaded neighboring part. Figure 1 explains the algorithm using three, 2D examples for clarity. The procedure has been implemented for 3D meshes. By this local inter-part boundary adjustment, the vertex imbalance is improved while only modestly perturbing the element balance. The procedure is repeated for several iterations to achieve the desired improvement to the vertex balance. In the extreme scale simulations, the global and local partitioning are used in a combined manner. i.e., in the first step, the mesh is balanced globally into an intermediate number of parts, \( m \), and in the second step, each of these \( m \) parts split independently to
Figure 1. Partition before and after LIIPBMod.

$n$ parts, which gives $m \times n$ parts in total. In the final step, the balance of the partition is improved by an iterative mesh entity migration procedure. This combination is much faster and more efficient compared to global partitioning.

An abdominal aorta aneurysm (AAA) model (see Figure 2) is used to study the partition improvement algorithm by using it in conjunction with graph/hypergraph-based procedures, namely ParMETIS PartKWay [9] and Zoltan Parallel HyperGraph partitioner PHG [7].

First consider a 1.07 billion anisotropic, tetrahedral element mesh created by applying mesh adaptation [10] on an AAA model. The global (ParMETIS PartKWay) and local (PHG) partitioning are combined to obtain different partitions with number of parts ranging from 4,096 to 294,912 [11]. The element imbalance is within 6% for all the partitions with up to 294,912 parts. However, the vertex imbalance is getting worse when the mesh is distributed to more and more parts. Each part of a globally balanced (1.025% element imbalance) partition with 4,096 parts splitting to 72 parts gives a partition with 294,912 parts in total. The element imbalance is 5.6%, but the vertex imbalance is 43.7%, which indicates 43.7% more work to do on the part with the highest number of vertices during the equation solution phase of the FEA. The LIIPBMod algorithm reduces the vertex imbalance dramatically to 17% while increasing the element imbalance to 15%. Table 1 presents the time usage of the FEA along with the scaling factors. It also compares the cases with and without using LIIPBMod algorithm. The equation solution phase is accelerated by 12% (from 10.38 to 9.14 seconds) due to better vertex balance, while the equation formation is slowed down a little from 5.54 to 5.82 seconds. The total time usage of the FEA is reduced from 15.92 to 14.96 seconds and the scaling factor is increased from 0.74 to 0.79. The time spent on the FEA is reduced by 0.96 second per 20 time steps, which means 78.6 cpu hours. In the real application, we usually run thousands of time steps per cardiac cycle, e.g., 5000. By using LIIPBMod algorithm, we save 78.6 × 5000 = 393

<table>
<thead>
<tr>
<th>1.07B element mesh</th>
<th>Eqn. form.</th>
<th>Eqn. soln.</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>num. of cores</td>
<td>Time</td>
<td>s_factor</td>
<td>Time</td>
</tr>
<tr>
<td>4,096 (base)</td>
<td>390.17</td>
<td>1</td>
<td>455.51</td>
</tr>
<tr>
<td>294,912</td>
<td>5.54</td>
<td>0.98</td>
<td>10.38</td>
</tr>
<tr>
<td>294,912 (LMod)</td>
<td>5.82</td>
<td>0.93</td>
<td>9.14</td>
</tr>
</tbody>
</table>

Figure 2. Geometry and mesh of an AAA model.
thousand cpu hours per cardiac cycle in analysis. The time spent on LIIPBMod algorithm is much smaller than the classical graph/hypergraph-based partitioner, which is negligible compared to solver [6]. An 8.56 billion element mesh was also run and yielded similar results.

3. Neighborhood Aware Message Packing
The Inter-Processor Communication Manager (IPComMan) is a package to reduce data exchange costs by exploiting communications of a local neighborhood for each processor. The neighborhood is the subset of processors exchanging messages with each other during a communication round, which in a number of applications is bounded by a constant, typically under 40, independent of the total number of processors. The library takes care of the message flow with a subset of MPI functions. The library provides several useful features (i) automatic message packing, (ii) management of sends and receives with non-blocking MPI functions, (iii) asynchronous behavior unless the other is specified, and (iv) support of dynamically changing neighborhood during communication steps. IPComMan takes care of memory allocation for both sending and receiving buffers, and manages the ones that can be reused without additional allocation. The library stores messages going to the same processor in contiguous memory. Thus, when sending or receiving a package, no additional memory copying is needed. The user may specify whether the size of each message is constant or arbitrary during a specific communication step. The fixed message size is taken by the library and used while extracting the messages. The arbitrary message size is put together with every message to correctly unpack the message upon arrival.

While initializing a communication library object, each processor specifies the neighbors it is going to communicate with. From that point, the library is concentrated on delivering messages between neighbors only, not touching other processors of the domain, when required. There are no collective calls during each communication round that remains within a neighborhood. There could be situations when it is not possible to a priori define all the neighbors for processors, i.e., new neighbors may be encountered during a communication step. For example, mesh modification operations may alter the neighborhood of specific processors. Consider the communication pattern presented in Figure 3. Processor P0 has in its neighborhood processors P1 and P2. P3 has P2 as the only neighbor, but after it has sent all the messages to P2 it finds out that there are some messages to be sent to P0. P3 includes P0 in its list of neighbours and begins to send packages to it. An increment of time before, say P0 finished sending to and receiving all the messages from P1 and P2, extracted them and proceeded to the next communication step. In that case packages from P3 to P0 would be lost, which will result in incorrect program behavior. To avoid this problem as new neighbors needing to communicate are encountered, one collective call at the end of communication round is performed to verify whether the global number of sends and receives match. When new neighbors are formed, the library continues to receive the messages identifying the new neighbors and communications. Note that at that time all the packages from existent neighbors are already received.

![Figure 3](image_url)

**Figure 3.** Communication paradigm with the neighborhood being changed.

Dynamic and irregular computation often results in an unpredictable number of messages communicated among the processors. Using IPComMan, there is no need to verify and send the
number of packages to be received at the end of sending phase. The last message sent from the
processor to a neighbor contains the number of buffers expected to be received by the neighbor.

To measure the performance of mesh migration, an essential part of parallel adaptive unstructured
mesh procedures [12], IPComMan was compared with the Autopack communication utility. Initial
results on IBM Blue Gene/L [3] show that IPComMans ability to localize communication to
neighborhoods to be independent of the total number of processors and its use of non-blocking
functions allows it to continuously reduce the communication time as the number of processors
increases. Even though the differences in communication times between Autopack [13] and
IPComMan are not substantial at 128 processors, IPComMan is from 3 to 7 times faster in the 4096
processor case.

Although the total communication time for IPComMan is reduced, the scaling for the process does
fall with increased processor count. It is important to note though that the mesh adaptation process
that involves mesh migration procedures is not well balanced in terms of the communication load per part.
However, additional efforts on the scalability are desired. This includes several aspects of managing
sends and receives during the communication phase while interacting with the computation part, and
keeping efficient use of the buffer memory. The options are being considered to eliminate all the
collective calls with the neighborhood concept in IPComMan, although additional analysis is required
to see if these approaches lead to the reduction of the communication time in oppose of using
collective call with the processor count growth.

4. Closing Remark
As the number of cores used in our parallel adaptive simulations continue to increase it is becoming
increasingly important to ensure that the tools used to adapt the mesh and load balance it for the next
set of solution steps are efficient and scale, while at the same time produce a partitioning of the mesh
that ensures the solver will scale well (preferably strong scaling). This paper has outlined two services
that have been recently to help meet these needs.

References
Numerical Mathematics, 26 241–263.
Computing, 23 1327–1347.
78-012053 012053.
(under review).
review.
[13] Loy R Autopack user manual science Division Argonne National Laboratory,
http://wwwunix.mcs.anl.gov/autopack/.