Parallel Data Refinement Layer of a Telescopic Approach for Extreme-scale Parallel Mesh Generation for CFD Applications

Nikos Chrisochoides, Andrey Chernikov, Thomas Kennedy, Christos Tsolakis, and Kevin Garner
CRTCLab, Computer Science Department
Old Dominion University

Abstract

In earlier work, we proposed a Telescopic approach which is a multi-layered approach for extreme-scale parallel mesh generation and adaptation. In this paper, we describe the Parallel Data Refinement (PDR) layer of the Telescopic approach. Namely focus on PDR's: (i) design and implementation and (ii) evaluation using TetGen, an open source mesh generation software, on shared memory machines. We outline lessons learned and future directions for revisiting the PDR layer and making adjustments in the implementation of the remaining layers of the Telescopic approach.

1. Introduction

Our long term goal is to achieve extreme-scale adaptive CFD simulations on the complex, heterogeneous High Performance Computing (HPC) platforms. To achieve this goal, we proposed in a telescopic approach (see Figure 1) [1] [2]. The telescopic approach is critical in leveraging the concurrency that exists at multiple levels in anisotropic and adaptive simulations. At the chip and node levels, the telescopic approach deploys a Parallel Optimistic (PO) layer and Parallel Data Refinement (PDR) layer, respectively (see Sections below). In this paper, we focus on the implementation of PDR. In future efforts, the PDR layer will be implemented on top of the PO layer [3]. The PDR layer relies on theory presented for parallel mesh generation and adaptivity [4] [5] [6]. The requirements for parallel mesh generation and adaptivity are:

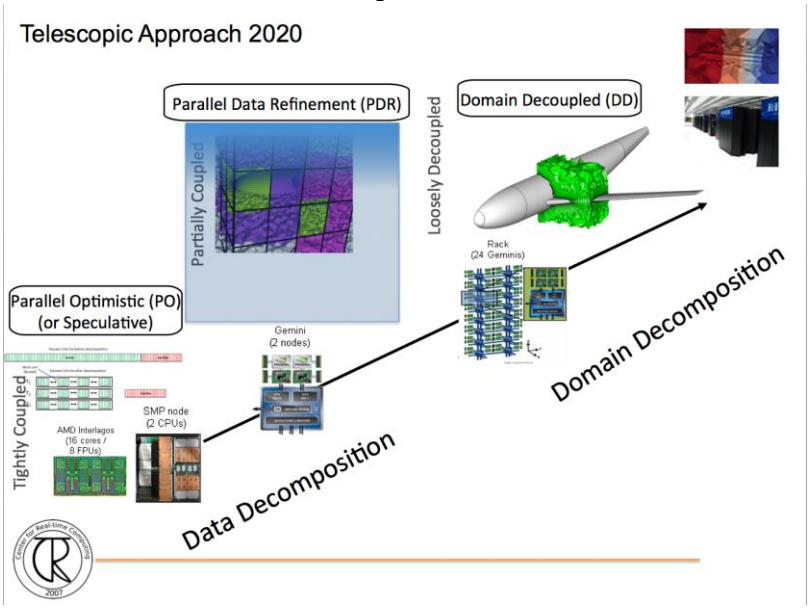


Figure 1 Telescopic Approach to parallel mesh generation and adaptation and PDR layer in the middle.

1. **Stability:** the quality of the mesh generated in parallel must be comparable to that of a mesh generated sequentially. The quality is defined in terms of the shape of the elements (using a chosen location/error-dependent metric), and the number of the elements (fewer is better for the same shape constraint).
2. **Robustness:** the ability of the software to correctly and efficiently process any input data. Operator intervention into a massively parallel computation is not only highly expensive, but most likely infeasible due to the large number of concurrently processed sub-problems.
3. **Code re-use:** a modular design of the parallel software that builds upon previously designed sequential meshing code, such that it can be replaced and/or updated with a minimal effort. Code re-use is feasible only if the code satisfies the reproducibility criterion, identified for the first time in this project. However, the experience from this project indicated that the complexity of state-of-the-art codes inhibits their modifications to a degree that their integration with parallel frameworks like PDR becomes impractical.
4. **Scalability:** the ratio of the time taken by the best sequential implementation to the time taken by the parallel implementation. The speedup is always limited by the inverse of the sequential fraction of the software, and therefore all non-trivial stages of the computation must be parallelized to leverage the current architectures with millions of cores.

The experience from the PDR implementation suggests that the code-reuse criterion ought to be adjusted for several reasons: (i) complexity of industrial strength sequential mesh generation codes and (ii) their reproducibility properties.

1.1 Reproducibility

A sequential mesh generation code like TetGen is an obvious choice for local mesh refinement within PDR. However, in distributed memory implementations (i.e., using MPI), a mesh needs to be reconstructed out of a set of points and tetrahedra in order to populate the data structures of TetGen and to then continue refining the subdomains.

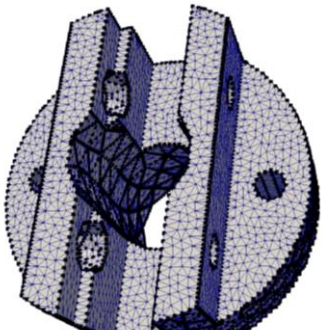


Figure 2. Coarse mesh of the socket geometry. Black dots represent points where TetGen reports errors

Although this capability exists as a function in TetGen (tetgenmesh::reconstruct()), the reconstruction procedure is not robust enough even for simple inputs. For example, for the socket geometry depicted in Figure 2 and using TetGen 1.5.0 [7] with an input of a set of points and tetrahedra produced from TetGen itself, the TetGen software reports many errors about wrong segment connections and then aborts. This type of behavior is exacerbated when more complex volumes generated from some data decomposition of a volume mesh is considered as input.

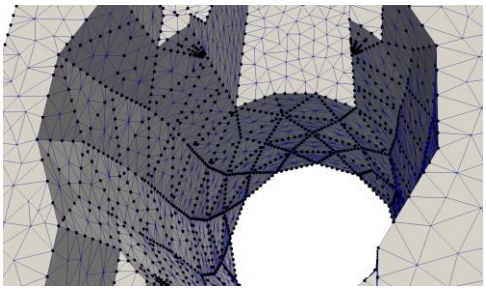


Figure 3. Zoom-in view of the center of the socket

This is a generic error due to wrong initialization in

some of the core data structures. TetGen provides functions that test the integrity of the internal data structures. Enabling these functions and visualizing the information reveals that at 2363 out of the 10472 boundary points (see Figure 3), the `tetgenmesh::reconstructmesh()` procedure did not link the segments appropriately. Even by passing more information to the aforementioned function, that is, boundary edge and boundary face connectivity, the final result is the same.

The reproducibility criterion is not required for shared memory PDR implementations, since the mesh is not reconstructed at any leaf during the refinement; instead, the PDR layer accesses the mesh directly. However, in the case of distributed memory implementations of the PDR layer, the original sequential software (e.g., TetGen) is initialized differently in the threaded versus the MPI versions. In the former one, the mesh is unique and available to all threads, which means that if one considers the boundary as a union of faces and segments it is unique throughout the execution. On the other hand, in the MPI version the parts of the mesh that are under refinement are oblivious to the rest of the mesh. This causes TetGen to treat the subdomain (in the case of PDR an octree leaf) boundary as an external boundary even if it lies away from the object's boundary, which means that special pre-refinement of edges and faces will occur. To avoid this, the TetGen code had to be explicitly modified by disabling these two pre-refinement stages.

2. Parallel Data Refinement (PDR) Algorithm

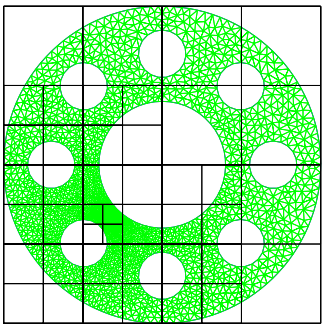


Figure 4. 2D octree for PDR

One way to decompose a meshing problem is through the introduction of permanent sub-domain boundaries to which the final mesh has to conform. However, any extra boundary adds constraints to the meshing problem, and therefore reduces the available optimization space. For example, one small angle or short segment can cause mesh quality deterioration on the global scale. The automatic construction of suitable three-dimensional subdomain boundaries that do not over-constrain the meshing problem is a very hard task that has not been solved to the team's knowledge. In fact, this is one of the major problems for the "pseudo" constrained AFLR modifications to be realized so far.

The PDR approach is instead based on data decomposition, illustrated in Figure 4, where the boundaries of the octree nodes do not become part of the mesh.

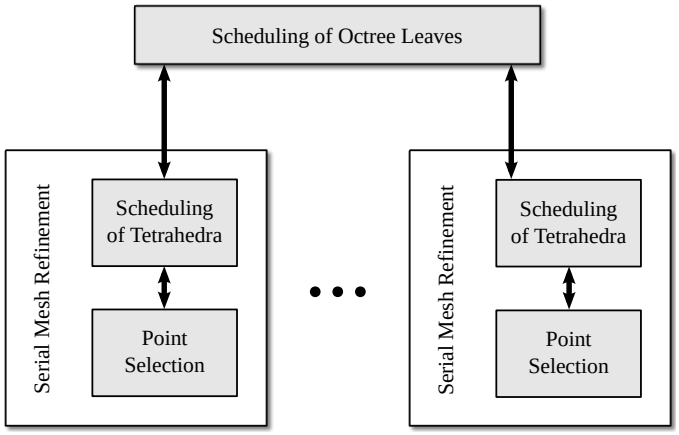


Figure 5. Software Architecture of PDR approach

Modular software design for code re-use. Figure 5 presents a high level diagram of the software design. The blocks

marked Serial Mesh Refinement represent P instances of a sequential refinement code, which is TetGen in the current implementation, and had been planned to be replaced by AFLR. The Element Scheduling boxes represent the management of poor quality element queues by the sequential code. In this implementation, TetGen's worklist was split to create a separate queue for each of the leaves of the octree. One schedules only one leaf at a time for refinement by a single core/thread, so that each thread draws from and pushes into a separate poor element queue. The Point Selection box is the abstraction for choosing a particular strategy for inserting points to eliminate poor quality tetrahedra. As shown previously [8], sequential Delaunay refinement algorithms have the flexibility to choose Steiner points from entire regions inside the circumspheres of poor quality tetrahedra, which are named selection balls. This approach is likely to extend to Advancing Front (AF) methods, too. The box marked "Scheduling of Octree Leaves" represents the construction of the octree and the scheduling of leaves for refinement. The leaves with larger volumes have higher refinement priorities than the leaves with smaller volumes, and the leaves of the same size are processed in first-in-first-out order. This strategy is designed to achieve high concurrency as early as possible in the progress of the algorithm without introducing large overheads.

Abstract Data Structures. If a part of the mesh associated with a leaf L of the octree is scheduled for refinement by a thread, no other thread can refine the parts of the mesh associated with the buffer zone BUF(L) of this leaf. This avoids fine grain synchronization overheads associated with concurrent point insertions. For each leaf L of the octree the following relation is maintained, which is required for the proof of correctness [6]: the circumradius of any tetrahedron which intersects L is less than 1/6 of the side length of L. In the case of AF methods, a similar argument can be made, but there is no mathematical proof due to the heuristic nature of AF methods.

When an octree leaf, L, is scheduled for refinement, one removes not only the nodes in the buffer zone BUF(L) from the refinement queue, but also the nodes from BUF(L') for each L' in BUF(L). Although this is not required by the theory, there are two implementation considerations for doing so, and both are related to the goal of reducing fine-grain synchronization. First, each leaf has an associated data structure that stores the poor quality tetrahedra whose circumspheres intersect this leaf, so that the circumradius-to-leaf side ratio can be maintained. Even though in theory the refinement of the mesh by concurrent threads is not going to cause problems when the threads work within the same octree leaf, in practice synchronization would need to be introduced for updating these poor quality lists. Second, for efficiency considerations, each tetrahedron contains pointers to neighboring tetrahedra for fast mesh traversal. However, if two cavities share an edge and are updated by concurrent threads, which can be done legitimately in certain cases, these tetrahedron-neighbor pointers will be invalidated. For these reasons, the sets of leaves affected by the mesh refinement performed by multiple threads were completely separated.

The PDR algorithm is designed for the execution by one master thread, which manages the work pool, and by multiple refinement threads that refine the mesh and the octree. The poor quality tetrahedra whose Steiner points are inside the square of L are stored in

the data structure denoted here as PoorTetrahedra(L). Leaf L needs to be scheduled for refinement if this data structure is not empty. In addition, each leaf has a counter for the tetrahedra that violate one half of the circumradius-to-leaf side ratio. When such a counter associated with L becomes zero, it implies that this ratio would hold for each of the children of L, and L can be split. As a result of a leaf subdivision, the overall concurrency is increased.

2.1 PDR Implementation

There are two parallel mesh generation approaches for the implementation of the PDR algorithm:

1. Given a surface, generate in parallel a volume mesh. A **progressive PDR** method exploits maximum concurrency as soon as it is feasible according to the PDR theory.
2. Given an initial volume mesh in core, adapt (refine) the mesh in parallel to meet error-based metrics. A **non-progressive PDR** method explores a fixed level of concurrency, in contrast to the progressive PDR method.

2.1.1 Progressive PDR Approach

High level description of the algorithm:

1. Input: surface mesh
2. Create a coarse size volume mesh (sequentially).
3. Create the root box of the octree.
4. Refine the mesh and the octree in parallel as needed, maintaining the circumradius-to-leaf size ratio.

By allowing the octree and the mesh to be refined simultaneously, a finer mesh can be constructed. Parallel refinement can progressively utilize additional cores as the available concurrency increases. The progressive approach reduces the startup overheads present in the non-progressive approach. The progressive approach introduces the need to classify certain tetrahedra as *obstacles* (to the splitting of octree leaves). A *potential obstacle* is a tetrahedron whose circumsphere intersects an octree leaf. One tetrahedron can be a potential obstacle to several octree leaves, and one octree leaf can have multiple potential obstacles. An *actual obstacle* to leaf L is any potential obstacle that violates the circumradius-to-leaf side ratio for L.

The progressive approach requires that each tetrahedron and its metadata be recorded. Each leaf contains a look-up table with records for all of its potential obstacles. This is necessary to redistribute the potential obstacles to the children of this leaf, and as a result some potential obstacles for L can become actual obstacles for the children of L. These tables are maintained by three operations: **(i)** Obstacle Registration**, (ii)** Obstacle Deregistration**, and (iii)** Distribution of Obstacle Entries. Obstacle Registration is the process during which a tetrahedron circumsphere is checked against the leaf currently under refinement and its first level neighbors. Each intersected leaf is recorded and added to a list. This list and the circumsphere are entered into the look-up table of each intersected leaf. The tetrahedron pointer serves as the key.

Obstacle Deregistration is the process during which a tetrahedron that has been split or updated is removed from all look-up tables. During this process, the list of intersected leaves is retrieved, by using the tetrahedron pointer to retrieve the corresponding entry from the look-up table of the leaf currently under refinement. Once the entry has been retrieved, it is removed from the look-up tables of all intersected octree leaves. Distribution of Obstacle Entries is the process during which obstacle entries are redistributed to the children of a leaf. This process occurs immediately after a leaf is split. Each entry in the look-up table for the leaf is reassigned to the corresponding child. The list of intersections is updated replacing the split node with the appropriate children.

The cost of obstacle tracking is mesher-dependent. The overhead of tracking obstacles was further reduced by attaching direct pointers to each tetrahedron instead of using a look-up table (as in the current implementation after properly modifying the pertinent internal tetrahedron data structure for TetGen 1.4).
Data Structures: The first two processes (Obstacle Registration and Deregistration) are dependent on the selected data structure for the look-up table implementation. The original look-up table implementation was based on the C++ std::map. This container is based on the sorting of the keys. The insert and erase std::map methods used during Obstacle Registration and Obstacle Deregistration respectively, are logarithmic in complexity. The current look-up table implementation is based on the C++11 std::unordered_map. The unordered_map::insert operation (used during Obstacle Registration) has a best case constant complexity and worst case linear complexity. The unordered_map::erase method has an average case constant complexity and worst case linear complexity. The results for Progressive PDR refinement show 70% parallel efficiency for about 800M elements on a 32-core node.

2.1.2 Non-Progressive PDR Approach

High level description of the algorithm:

1. Input: surface mesh
2. Create a volume mesh. Create an octree of depth N.
3. Refine the volume mesh sequentially to match the circumradius-to-leaf side ratio for all octree leaves.
4. Refine the volume mesh in parallel

An Octree of depth 4 provides sufficient concurrency for a 32-core node and initial mesh of 703K elements (it takes 21 sec to generate). A higher depth tree implies a higher initialization cost and thus the need for progressive PDR. A larger octree (i.e., expectations for higher concurrency) requires a larger size mesh. The benefit of the non-progressive approach is in avoiding the overheads associated with the maintenance of the obstacles. This is done by a single call to TetGen instructing it to refine all tetrahedra below the specified circumradius bound.

3. Performance Evaluation

PDR.Tetgen was compiled on CentOS with version 4.9 of the gnu g++ compiler. Local analysis was performed on the ODU Turing Cluster on a high memory node (32 cores

and 768 GiB RAM). PDR was evaluated for up to 32 cores. Analysis of large meshes (>500M elements) was performed on the retired PSC Greenfield machine. Greenfield was a large Distributed Shared Memory (DSM) machine; with each node contained 4 15-core CPUs and provided 3 TiB RAM. PDR was evaluated for up to 75 cores.

3.1 Quantitative Results

This section presents preliminary performance data collected to identify pros and cons of the two different PDR approaches: (i) progressive and (ii) non-progressive, using three different size meshes: (i) small, (ii) medium, and (iii) large size meshes.
Table 1 presents observations with respect to two different approaches and meshes of various sizes.

Table 1. Overview of Progressive PDR Behavior using TetGen.

Mesh Size	Parallel Mesh Generation	Adaptability
Small (2M to 100M)	Progressive performs worse than Non-Progressive PDR	High startup time
Medium (>100M)	Progressive performs better than Non-Progressive PDR for more than 4 cores	Minimal startup time (for appropriate initial octree depth)
Large (>500M)	Progressive has higher runtime than Non-Progressive PDR. Lower idle times within worker threads.	Minimal startup cost (for appropriate initial octree depth)

3.1.1 Progressive PDR allows the initial octree depth to be tuned to the number of available cores. By decoupling the initial octree depth and the final octree depth, an initial subset of available cores can be utilized. As octree leaves are split, additional work becomes available, and additional cores can be utilized.

Remark: The progressive PDR approach has a very low cost for the initial mesh construction, since it progressively utilizes cores on demand, i.e., uses as many cores as the concurrency in the computation becomes apparent. This approach is suitable for parallel mesh generation (Problem I) when the input is a surface mesh or comes from a CAD model. Table 2 indicates that the scaled (weak) speedup of the method is very good.

Table 2. Runtime breakdown for Progressive PDR for selected medium meshes (approximately 2 million to 70 million elements). Time (in seconds) is divided into time spent in each phase of PDR.

# Cores	1	2	4	8	16	32
Total	84.28	87.51	90.84	105.16	125.63	226.82
Octree Construction	3.49	3.38	3.53	8.69	8.15	7.44
Mesh Construction	2.63	2.66	2.65	21.92	22.19	20.94
Parallel Refinement	78.15	81.47	84.65	74.51	95.25	198.38
# Elements	1,890,949	3,682,307	6,418,835	12,532,332	29,537,735	70,683,194

3.1.2 Non-Progressive PDR constructs the final octree before parallel refinement begins. If non-progressive PDR requires an octree of depth *N*, the octree must be constructed for a given initial mesh, i.e., only the final octree depth can be tuned. This approach is suitable for parallel adaptive mesh refinement (Problem II) when the input is a volume mesh already generated in parallel (e.g., using the progressive PDR approach) and requires mesh refinement. Consequently, by eliminating the initial mesh construction in Table 1, one observes good scalability for the non-progressive PDR method even for small & medium size meshes. While, Table 3 lists a breakdown of the runtimes for meshes of selected sizes, where the initial mesh construction is removed. The octree construction and initialization is taking place sequentially; this is an area for improvement.

Table 3. Runtime breakdown for Non-Progressive PDR for medium meshes (approximately 2 million to 70 million elements). Time (s) is divided into time spent in each phase of PDR. Octree Construction and Mesh Construction are sequential operations. For 1 and 2 cores an octree of depth 4 is constructed, while for 4 and higher number of cores and octree of depth 5 is constructed.

# Cores	1	2	4	8	16	32
Total	32.72	36.33	75.25	40.54	50.19	64.37
Octree Construction	4.12	3.82	16.43	16.26	16.23	16.49
Parallel Refinement	28.60	32.75	58.77	24.08	33.96	47.88
# Elements	1,995,089	3,718,464	7,281,459	13,694,447	29,762,341	70,192,796

Remark: The overheads introduced by obstacles and neighbor re-computation cause Progressive PDR to perform worse than Non-Progressive PDR. However, one needs to take into account that the progressive PDR generates both the octree and the initial mesh.

In summary, we observe good scalability for a small (< 32) number of cores for PDR. However, different PDR approaches behave differently for small to medium size meshes. Namely, (i) **progressive PDR** is suitable for parallel mesh generation that starts from CAD (in the present study a surface mesh) and generates a volume mesh. Current end-to-end times for a generated mesh of size 796.4M tets, using different numbers of cores are: 1 core: 26,122 sec, 16 cores: 2064 sec, and 32 cores: 1596 sec. These data suggest a fixed speedup of more than 16. There are still a number of optimizations that would help improve upon this performance; (ii) **non-Progressive PDR** is suitable for adaptive mesh refinement of an existing mesh. Current end-to-end time (including the cost for an initial coarse mesh) for a generated mesh of size 797M tets, using different numbers of cores are: 1 core: 17,010 sec, 16 cores: 1,646 sec, 32 cores: 1,633 sec. The optimum end-to-end performance is for 24 cores: 1,560 sec. These data suggest a fixed speedup of about 11, when the time for the initial coarse mesh is included. However, the non-progressive method will be used in the context of adaptive volume mesh refinement where the mesh to be refined concurrently is already generated and the cost is charged in an earlier phase.

3.2 Qualitative Evaluation

Figure 6 shows the angle distributions for the socket geometry using Non-Progressive PDR on a single core (a) and Non-Progressive PDR on 32 cores (b). While, Figure 7 shows the face angle distributions for the socket geometry for Non-Progressive PDR for a single core (a), and Non-Progressive PDR for 32 cores (b). Finally, Figure 8 shows the

aspect ratio distributions for the socket geometry for Non-Progressive PDR for a single core (a), and Non-Progressive PDR for 32 cores (b) using TetGen 1.4.

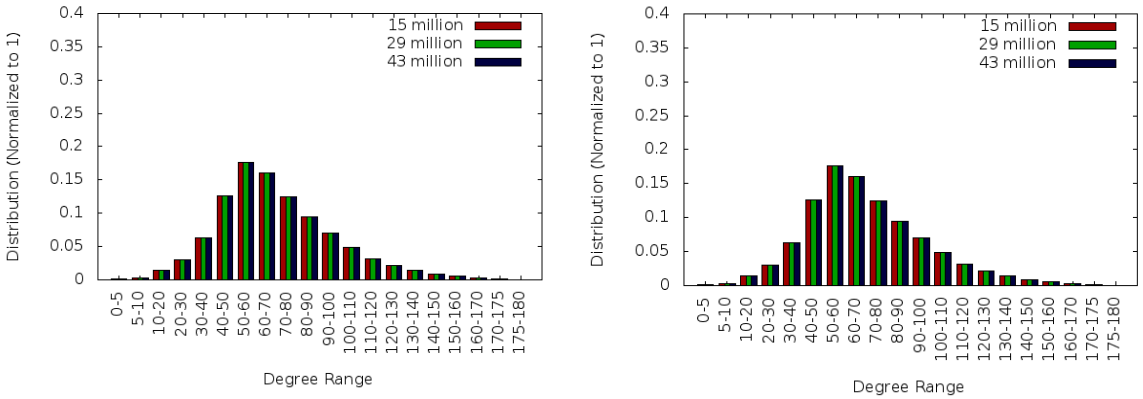


Figure 6. Dihedral angle distributions for the socket geometry. The y-axis runs from 0 to 0.4. On the left (a) the dihedral angle distribution for Non-Progressive PDR when run on a single core is shown. On the right (b) the distribution for Non-Progressive PDR on 32 cores is shown.

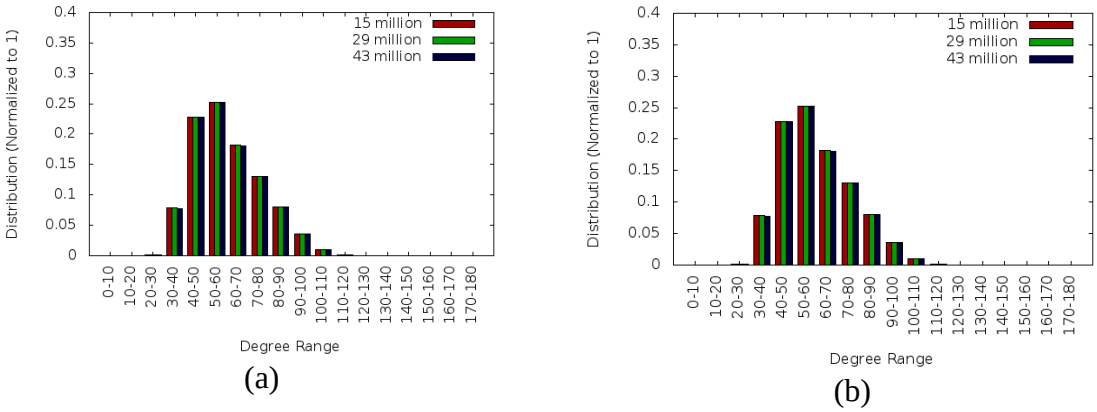


Figure 7. Face angle distributions for the socket geometry. The y-axis runs from 0 to 0.4. On the left (a) the face angle distribution for Non-Progressive PDR when run on a single core is shown. On the right (b) the distribution for Non-Progressive PDR on 32 cores is shown.

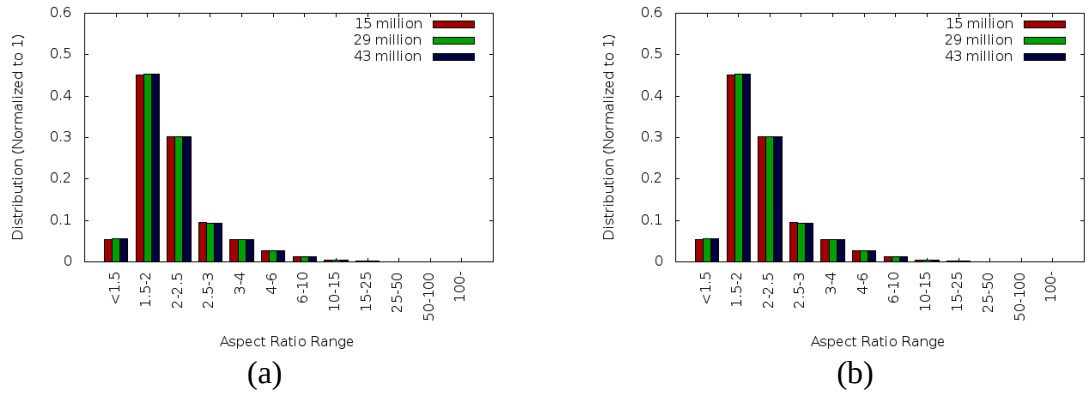


Figure 8. Aspect ratio distributions for the socket geometry. The y-axis runs from 0 to 0.6. On the left (a) the aspect ratio distribution for Non-Progressive PDR when run on a single core is shown. On the right (b) the distribution for Non-Progressive PDR on 32 cores is shown.

The stability and robustness of the PDR method for parallel isotropic Delaunay-based mesh generation is not only theoretically proven [4] [5] [6], but experimentally verified. Similar behavior is observed for the Advancing Front type of methods. Figure 9 depicts a defroster geometry and Figure 10 depicts a curved duct geometry.

Figure 11 shows qualitative results for the refinement of these two geometries using an early implementation of PDR with AFLR. Detailed description of these efforts will appear elsewhere.

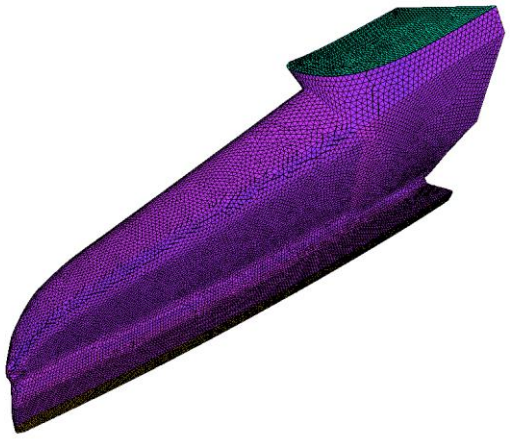


Figure 9. Defroster Geometry

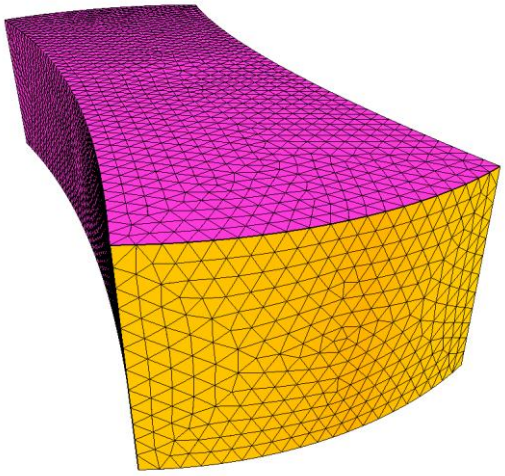
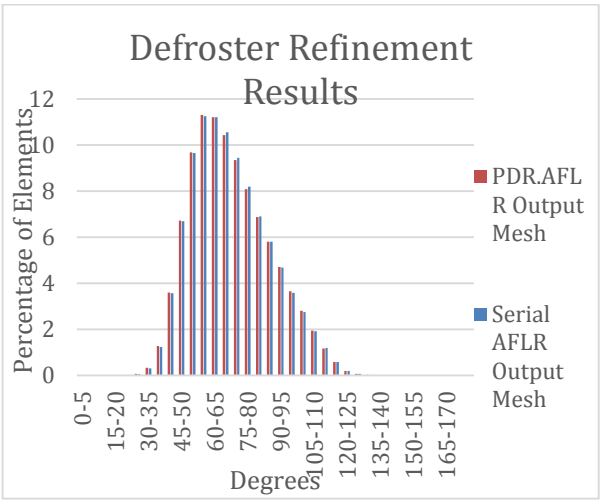
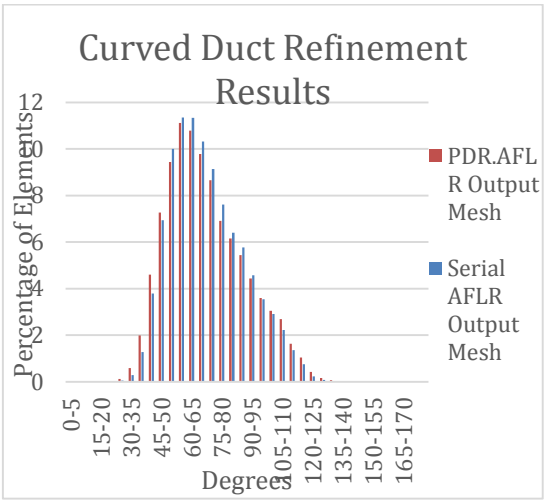


Figure 10. Curved Duct Geometry



(a)



(b)

Figure 11. Dihedral angle distributions of the output meshes are shown and compared between the serial AFLR code and PDR.AFLR. The left (a) gives the qualitative results of the defroster geometry and the right (b) gives the results of the curved duct geometry.

4. Conclusions and Future Work

The lessons learned from the challenges faced with the implementation of the PDR layer suggest:

- The complexity of existing state-of-the-art sequential mesh generation codes for building robust HPC parallel codes is very high – to a degree that their modifications can NOT even be managed by their own original creators within reasonable time-frames and even for the simplest of the parallel mesh generation approaches (i.e., PDR-type methods); their main challenge is to achieve robustness and stability.
- In the long run, the CFD community is better off building from scratch new parallel mesh generation codes such as CTD3D [3] implemented as an alternative to TetGen and AFLR.

Due to TetGen's lack of reproducibility (AFLR meets the weak reproducibility criterion), future efforts will be focused on the Distributed Shared Memory (DSM) implementation of the PDR framework. This will permit the efficient implementations of the Parallel Constrained (PC) step of the Telescopic approach. Because of reproducibility and software complexity issues that prevent major and proper modifications (especially in the case of fine grain optimistic layer of the telescopic approach) of state-of-the-art sequential software required for parallel mesh generation, at this point it is our experience that it is best to consider the design and implementation for new parallel mesh generation codes on small numbers of cores (i.e., multi-core shared memory machines) ready to be integrated in subsequent layers of the telescopic approach to achieve large- to extreme-scale codes.

Finally, early preliminary results suggest that the PDR layer is **suitable for end-user productivity** in the short term and it can likely be used within a multi-core single node platform to increase the concurrency of the Optimistic Layer. Preliminary (un-optimized) data on Distributed Shared Memory (DSM) machines suggest that at the node level, linear speedup up to 60 or 100 cores can be achieved. Figure 12 depicts the *fixed speedup remains close to 16* for a mesh with 800M element and *scalable (weak) speedup* is close to *20*. However, more work remains to be done to optimize the PDR scheduler for DSM nodes.

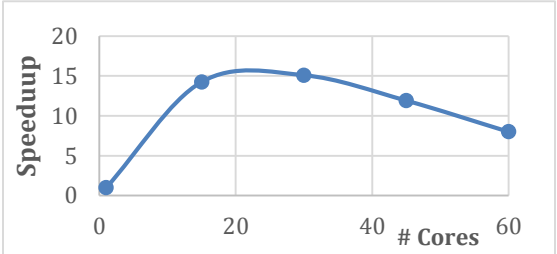


Figure 12. Scalable Speedup on DSM machine.

Acknowledgements This work in part is funded by NSF grant no. CCF-1439079, NASA grant no. NNX15AU39A and DoD's PETTT Project PP-CFD-KY07-007, Richard T. Cheng Endowment and M&S Fellowship at ODU. In addition, it utilized resources from the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575. The content is solely the responsibility of the authors and does not necessarily represent the official views of the government agencies that support this work. The authors thank Dr. David Marcum for the helping answer all the questions regarding the modifications or AFLR at ODU. The presentation of this document is improved substantially form the comments of K. Vogiatzis, H. Thornburg, and C. Peavey at Engility.

References

[1] N. Chrisochoides, "Telescopic Approach for Extreme-scale Parallel Mesh Generation for CFD Applications," in *AIAA Aviation*, 2016.

[2] N. Chrisochoides, A. Chernikov, A. Fedorov, A. Kot, L. Linardakis, and P. Foteinos, "Towards Exascale Parallel Delaunay Mesh Generation," in *Proceedings of the 18th International Meshing Roundtable*, Salt Lake City, UT, 2009.

[3] F. Drakopoulos, C. Tsolakis and N. Chrisochoides, "CDT3D : Fine-Grained parallel grid generator for Aerospace Applications," in *AIAA Aviation*, 2017.

[4] A. Chernikov and N. Chrisochoides, "Practical and Efficient Point Insertion Scheduling Method for Parallel Guaranteed Quality Delaunay Refinement," in *18th ACM International Conference on Supercomputing*, Saint-Malo, France, 2004.

[5] A. Chernikov and N. Chrisochoides, "Parallel Guaranteed Quality Delaunay Uniform Mesh Refinement," *SIAM Journal on Scientific Computing,* vol. 28, no. 5, pp. 1907-1926, 2006.

[6] A. N. Chernikov and N. P. Chrisochoides, "Three-dimensional Delaunay Refinement for Multi-core Processors," in *ICS '08*, Island of Kos, Greece, 2008.

[7] H. Si, "TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator," *ACM Trans. on Mathematical Software (TOMS)*, p. 36, 2015.

[8] A. N. Chernikov and N. . P. Chrisochoides, "Generalized Insertion Region Guides for Delaunay Mesh Refinement," *SIAM Journal on Scientific Computing*, vol. 34, no. 3, pp. A1333-A1350, 2012.