PARALLEL REFINEMENT OF TETRAHEDRAL MESHES USING TERMINAL-EDGE BISECTION ALGORITHM

Maria-Cecilia Rivara[1*] Daniel Pizarro[1†] Nikos Chrisochoides[2‡]

[1] *Universidad de Chile, Chile {mcrivara, dpizarro}@dcc.uchile.cl*
[2] *College of William and Mary, VA, U.S.A. nikos@cs.wm.edu*

ABSTRACT

In this paper we present a practical and simple terminal-edge algorithm for parallel refinement of tetrahedral meshes based on the refinement of terminal-edges and associated terminal stars. A terminal-edge is a special edge in the mesh which is the longest edge of every element that shares such an edge, while the set of elements that share a terminal-edge forms a terminal star. We prove that our simple algorithm makes implicit use of the same concepts and ideas than the more complex Lepp-bisection refinement algorithm, which in turn implies that far more points are inserted in the interior of the submeshes than in their interfaces. Empirical results illustrating its performance, stability and scalability are also included.

Keywords: parallel mesh generation, longest-edge, terminal-edge, Lepp

1. INTRODUCTION

Parallel mesh generation methods should satisfy the following four practical criteria: (1) *stability* in order to guarantee termination and good quality of elements for parallel finite element methods, (2) *simple domain decomposition* in order to reduce unnecessary pre-processing overheads, (3) *code re-use* in order to benefit from fully functional, highly optimized, and fine tuned sequential codes, and (4) *scalability*. Our parallel mesh generation algorithm satisfies the first two requirements, it has high code re-use and scales well for up to 64 processors. Although, theoretically it is not scalable for very large number of processors, our experimental data from up to **64** processors suggest that the central processor that maintains the global name space of vertices is not a major bottleneck. This suggests that our method is suitable for shared memory parallel machines and medium size Clusters of Workstations (CoWs) like the one we used here.

Parallel mesh generation procedures in general overdecompose the original mesh generation problem into N smaller subproblems which are meshed concurrently using P ($\ll N$) processors [8]. The subproblems can be formulated to be either tightly [26, 6] or partially coupled [21, 12, 7] or even decoupled [3, 34, 19]. The coupling of the subproblems determines the intensity of the communication and the degree of dependency (or synchronization) between the subproblems. The parallel mesh generation and refinement method we present in this paper is *an almost decoupled* method (i.e., there is no communication between the subproblems, but a central processor is required to maintain the global namespace of mesh points, for consistency).

There are two classes of parallel tetrahedral mesh generation methods: (1) Delaunay and (2) non-Delaunay. There are very few practical [13, 36, 7, 26, 19] methods on the parallelization of Delaunay mesh generation. Moreover, due to the inherent complexity of the 3-dimensional Delaunay algorithm there are (to

* *This work was partially supported by Fondecyt 1040713 and Foundation Andes C-13840*

† *This author worked on this method during his visit at College of William and Mary, and he was supported in part by Foundation Andes, Chile (C-13840 - 2003/2004)*

‡ *This author's work is supported in part by NSF Career Award #CCR-0049086, ITR #ACI-0085969, NGS #ANI-0203974, and ITR #CNS-0312980*

the best of our knowledge) only two practical parallel guaranteed quality Delaunay mesh generation algorithm [13, 26] for polyhedral domains and one for general domains [13]. The algorithm in [13] starts by sequentially meshing the external surfaces of the geometry and by pre-computing domain separators whose facets are Delaunay-admissible (i.e., the precomputed interface faces of the separators will appear in the final Delaunay mesh). The separators decompose the continuous domain into subdomains which are meshed in parallel using a sequential Delaunay mesh generation method on each processor. The method is stable.

The algorithm in [26, 6] maintains the stability of the mesher by simultaneously partitioning and refining the interface surfaces and volume of the subdomains [10] —a refinement due to a point insertion might extend across subproblem (or subdomain) boundaries (or interfaces). The extension of a cavity beyond subdomain interfaces is a source of irregular and intensive communication with variable and unpredictable patterns. Although the method in [26, 6] can tolerate up to 90% of the communication—by concurrently refining other regions of the subdomain while it waits for remote data to arrive—its scalability is of the order of $O(\log P)$, P is the number of processors. Unfortunately, the concurrent refinement can lead to a non-conforming mesh and/or non-Delaunay mesh [26, 6]. These non-conformities are resolved at the cost of setbacks which require algorithm/code re-structuring [11] or at the cost of complete re-triangulation of the global mesh [35] each time a set of independent points is inserted [37].

On the other hand, longest-edge bisection algorithms, introduced by Rivara [28, 29, 31], appear to be more popular for the refinement/derefinement of triangulations for adaptive finite element methods both in the serial and parallel settings [23, 25, 33, 38, 17, 5]. The algorithms are based on the bisection of triangles/tetrahedra by its longest-edge as follows: in two-dimensions this is performed by adding an edge defined by the longest-edge midpoint and its opposite vertex, while in three dimensions the tetrahedron is bisected by adding a triangle defined by the longest-edge midpoint and its two opposite vertices.

Jones and Plassman in [17] have proposed a two-dimensional, 4-triangles parallel algorithm for the refinement / derefinement of triangulations, which in order to avoid synchronization tasks uses a Monte Carlo rule to determine a sequence of independent sets of triangles which are refined in parallel. The algorithm also takes into account additional triangles to be refined to obtain a conforming mesh. In order to minimize the latency and communication costs, a mesh partitioning algorithm based on an imbalanced recursive bisection strategy is also used.

Castaños and Savage in [5] have parallelized the non-conforming longest edge bisection algorithm (illustrated in Figure 1 in 2-dimensions) both in 2 and 3 dimensions. In this case the refinement propagation implies the creation of sequences of non-conforming edges that can cross several submeshes involving several processors. This also means the creation of non-conforming interface edges which is particularly complex to deal with in 3-dimensions. To perform this task each processor P_i iterates between a no-communication phase (where refinement propagation between processors is delayed) and an interprocessor communication phase. Different processors can be in different phases during the refinement process, their termination is coordinated by a central processor P_0. Duplicated vertices can be created by the non conforming interface edges. A remote cross reference of newly created interface vertices during the interprocessor communication phase along with the concept of *nested elements* [4] guarantees the assignment of the same logical name for these vertices. The load balancing problem is addressed by using mesh repartitioning based on an incremental partitioning heuristic.

In this paper we propose a non-Delaunay mesh tetrahedral refinement based on a local terminal-edge (and terminal-star) refinement operation instead of longest edge. Contrary to the method in [5] the new terminal-star refinement algorithm completely avoids the management of non-conforming edges both in the interior of the submeshes and in the inter-subdomain interface. This eliminates the communication between subdomains and thus processors. Similarly to Castaños et al. we use a single processor as coordinator of the global name space for all mesh points (vertices). However, we augment the role of the coordinating processor in order to implement a refinement algorithm that is based on a new and simpler edge-size density function. The midpoint vertex of each interface terminal-edge is assigned a unique global name (as soon as a processor requests its split) which allows the free and independent processing of the corresponding distributed terminal-star. Also, load balance is effectively achieved by using an overdecomposition and the runtime system we presented in [1, 2].

2. BACKGROUND

In two-dimensions the longest-edge bisection algorithm essentially guarantees the construction of refined, nested and unstructured conforming triangulations (where the intersection of pairs of neighbor triangles is either a common vertex, or a common edge) of analogous quality as the input triangulation. More specifically the repetitive use of the algorithms produce triangulations such that: (a) The smallest angle α_t of any triangle t obtained throughout this process,

satisfies that $\alpha_t \geq \alpha_0/2$, where α_0 is the smallest angle of the initial triangulation. (b) A finite number of similarly distinct triangles is generated in the process. (c) For any conforming triangulation the percentage of bad quality triangles diminishes as the refinement proceeds. Even when analogous properties have not been fully proved in three dimensions yet, both empirical evidence [31, 33] and mathematical results on the finite number of similar tetrahedra generated over a set of tetrahedra [14] allow to conjecture that these results are also valid in general in the three-dimensional setting.

More recently, the use of two new and related mathematical concepts - the longest-edge propagation path (Lepp) of a triangle t and its associated terminal - edge, have allowed the development of improved Lepp based algorithms for the longest edge refinement / derefinement of triangulations both in 2 and 3-dimensions [30, 22, 32]. Moreover, the application of Lepp / terminal edge concepts to the Delaunay context have also allowed the development of algorithms for the quality triangulation of PSLG geometries [22], for the improvement of obtuse triangulations [15, 24], and for approximate quality triangulation [27].

Either for improving or refining a mesh, the Lepp based algorithms use a terminal-edge point selection criterion as follows. For any target element to be improved or refined, a Lepp searching method is used for finding the midpoint of an associated terminal-edge which is selected for point insertion. Each terminal-edge is a special edge in the mesh which is the common longest edge of all the elements (triangles or tetrahedra) that share this terminal-edge in the mesh. Once the point is selected, this is inserted in the mesh. In the case of the terminal-edge refinement algorithm, this is done by longest-edge bisection of all the elements that share the terminal-edge, which is a very local operation and simplifies both the algorithm implementation and its parallelization. The process is repeated until the target element is destroyed in the mesh.

2.1 Longest-edge Vs. Terminal-edge Algorithms in 2-dimensions

The serial pure longest-edge bisection algorithm of Rivara [28], basically works as follows: for any target triangle t to be refined both the longest-edge bisection of t and the longest-edge bisection of some longest-edge neighbors is performed in order to produce a conforming triangulation. This task usually involves the management of sequences of intermediate non conforming points throughout the process as illustrated in Figure 1 for the refinement of target triangle ABC; note that the non conforming points in triangulation (a) were numbered in their order of creation. Figure 1(b) shows the final conforming triangulation.

Alternative longest-edge based algorithms (i.e. the 4-triangles longest-edge algorithm), which use a fixed number of partition patterns have been also proposed in [28, 23]. The 4-triangles algorithms maintain only one non-conforming vertex as the refinement propagates toward larger triangles; however it generalization to 3-dimensions is rather cumbersome.

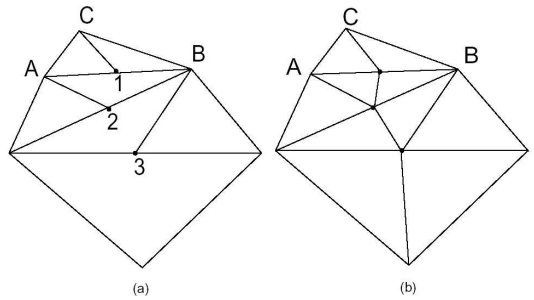

Figure 1: Refinement of triangle ABC by using previous longest edge refinement algorithm. (a) Intermediate non conforming triangulation (b) Final refined triangulation

The introduction of the following concepts has allowed the reformulation of the pure longest-edge algorithm:

Definition 1 *An edge E in any valid triangulation M is a terminal-edge if E is the longest edge of every triangle that shares such an edge. In addition the (one or two) triangles that share E are called terminal-triangles. In the case that E is an interior edge shared by a pair of terminal triangles (t_1, t_2) we shall say that they define a terminal quadrilateral $Q(t_1, t_2)$.*

Definition 2 *Let t_o be any triangle in M. The longest edge propagation path of t_0 (Lepp(t_0)) is the ordered list of triangles $(t_0, t_1, ...t_n)$ such that:*

(a) t_{i+1} is the neighbor of t_i by the longest edge of t_i for i=0, ..., n-1

(b) longest edge (t_{i+1}) > longest edge (t_i) for i=0,1,... N, where either N=n-1 and t_n has a boundary longest edge, or N=n-2 and (t_{n-1}, t_n) is a pair of terminal triangles

Note that, the Lepp(t_0) corresponds to a subtriangulation of an associated Lepp polygon, which captures the local point distribution around t_0 in the direction of its longest edge.

Proposition 1 *The terminal-edge associated to any Lepp(t_0) is the longest edge between all the edges involved in the subtriangulation Lepp(t_0) including the boundary of the Lepp polygon.*

Proof. The proof relies in the Lepp definition which involves finding a set of increasing longest edge triangles, which in turn define an associated Lepp polygon. The terminal edge is the last longest edge in the sequence, which is consequently longer than every edge of the involved triangles both in the interior and the boundary of the Lepp polygon.

An explicit Lepp based algorithm for the quality refinement of any triangulation was introduced in [30], where the refinement of a target triangle t_0 essentially means the repetitive longest-edge bisection of pairs of terminal triangles sharing the terminal-edge associated with the current Lepp(t_0), until the triangle t_0 itself is partitioned. For an illustration see Figure 2, where Lepp(t_0)=$\{t_0, t_1, t_2, t_3\}$ over the triangulation (a), and triangulations (b), (c) and (d) respectively illustrate the first, an intermediate and the last steps in the Lepp Bisection procedure. Note that the new vertices were enumerated in the order they were created. The generalization of this algorithm to 3-dimensions in formulated in next section.

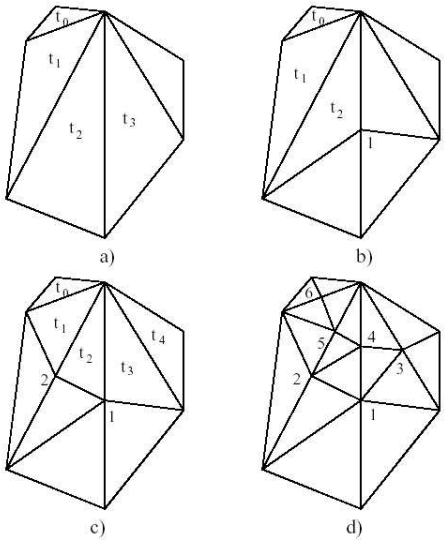

Figure 2: Lepp refinement of target triangle t_0

Note that Lepp / terminal edge algorithms use a very local and conforming mesh refinement operation which simplifies both the algorithm implementation and its parallelization in 2 and 3-dimensions.

2.2 Serial Lepp / terminal-edge Algorithms in 3-dimensions

As discussed in [30, 32], the 3-dimensional algorithm implies a multi-directional Lepp searching task, involving a set of terminal-edges. In this case each terminal-edge in the mesh is the common longest-edge of every tetrahedron that shares such an edge; and the refinement operation involves a terminal-star (the set of tetrahedra that share a terminal-edge) refinement. Note that the refinement is confined in the interior of the terminal-star.

Definition 3 *An edge E in any valid tetrahedral mesh M is a terminal-edge if E is the longest edge of every tetrahedron that shares E. In addition the set of tetrahedra that share E define a terminal-star in 3-dimensions, and every tetrahedron in a terminal star is called a terminal tetrahedron.*

Definition 4 *For any tetrahedron t_0 in M, the Lepp (t_0) is a 3-dimensional submesh (a set of contiguous tetrahedra) recursively defined as follows:*

(a) Lepp (t_0) contains every tethatedron t that shares the longest edge of t_0 and such that longest edge (t) > longest edge (t_0).

(b) For any tetrahedron t' in Lepp(t_0), the submesh Lepp (t_0) also contains every tetrahedron t not contained yet in Lepp (t_0) and such that t shares the longest edge of t' and longest edge (t) > longest edge (t').

Proposition 2 *In 3-dimensions, for any tetrahedron t in M, Lepp (t) has a finite, variable number of associated terminal-edges.*

Proof The proof follows from the fact that every tetrahedron t in any Lepp (t_0) has a finite, non fixed number of neighbor tetrahedra sharing the longest edge of t; and in the general case more than one of these tetrahedra has longest edge greater than the longest edge of t, which implies that the searching task involved in Definition 4 is multidirectional, and stops when a finite number of terminal edges, which are local longest edges in the mesh, are found in M.

A high level 3-dimensional refinement algorithm for the local refinement of a tetrahedral mesh, which extends the 2-dimensional algorithm illustrated in Figure 2 follows:

3D Lepp Bisection Refinement Algorithm
Input { a mesh M and set S of tetrahedra to be refined in M}
for each t in S **do**
  while t remains in M **do**

Find Lepp (t) and associated set of terminal edges (TE)
Refine each terminal star associated to every terminal edge in TE
end while
end for
Output {mesh M}

In this paper we focus on the parallelization of a global terminal edge refinement algorithm which makes implicit use of the Lepp concept. This serial algorithm performs the repetitive refinement of every terminal edge in the mesh greater than a given tolerance on the size of the terminal edges as follows:

Terminal-edge Refinement Algorithm
Input {a mesh M and a tolerance parameter b(M)}
Perform successive refinement of the terminal stars associated to terminal edges greater than b(M) in M until no terminal edge greater than b(M) remains in the mesh
Output {refined M}

3. PARALLEL CONSTRAINED TERMINAL-EDGE BISECTION ALGORITHM

In this section we consider a simple parallel terminal-edge bisection algorithm able to perform global refinement of tetrahedral meshes as follows: over each submesh, the parallel refinement of terminal-edges (and terminal-stars) is performed until the terminal-edge size is less than or equal to a global edge-size tolerance *b(M)*. A high level description of the method is:

Parallel Refinement Algorithm
1. *Read Input* {a mesh M and global edge-size tolerance b(M)},
2. *Partition the mesh M* in submeshes $M_i, i = 1...N$,
3. Distribute the submeshes M_i among the processors, and
4. *Perform constrained refinement over each* M_i.
5. Output

Note that the mesh mesh $M = M(D, V)$, with associated surface mesh D and associated set of vertices V, is partitioned into N submeshes $M_1(D_1, V_1), M_2(D_2, V_2), ..., M_N(D_N, V_N)$ and distributed to P processors.

During the parallel refinement, a coordinator processor assigns a unique global identifier (UGI) to each new vertex created in each submesh M_i.

The refinement over each submesh M_i is performed as follows:

Constrained Refinement over M_i
while there exists interior or interface terminal edges > b in M_i **do**
Step1. Perform repetitive refinement of every interior terminal edge greater than b in the interior of M_i and request a UGI for all new vertices
Step2. Perform repetitive refinement of every interface terminal edge greater than b in D_i and request their UGI
end while

It is worth noting that step1 (step2) in the while loop finishes when no more interior (interface) terminal edges are found. On the other hand, when the while is completed, no more (interior or interface) terminal edges greater than b remain in M_i.

In order to maintain a global name for all new vertices in M_i an UGI must be assigned to every new vertex created over D_i

Termination The refined mesh $M'(D', V')$ is constructed by simply computing the union of all of the D_i and V_i sets. Because of the UGIs, the resulting mesh will b conforming with no duplicate vertices on the submeshes interfaces.

3.1 Global Assignment of Vertices

The coordinator processor P_0 keeps record of the last UGI assigned. This record is incremented successively as new UGIs are assigned to processors P_i. P_0 acts like a ticket dispenser, where every ticket number is unique, and the ticket numbers are dispensed in ascending order.

Every time that a processor needs to bisect an interface edge in D_i, this sends a request for a UGI to P_0. If the edge was previously bisected by another processor, it will return the UGI value assigned to its midpoint. Otherwise, it will create a new UGI for the vertex generated on that edge.

3.2 Theoretical Framework

The following lemma assures that the parallel terminal edge algorithm deals with almost decoupled submeshes.

Lemma 3.1 *The use of a global edge-size tolerance parameter b(M) completely avoids interprocessor communication*

Proof Since a coordinator processor globally deals with the information on b(M), no information needs to be communicated between processors in order to refine interface terminal edges.

The following results based on the Lepp properties, assure that most of the refinement task is performed in the interior of the submeshes M_i.

Lemma 3.2 *Let E be any interior terminal edge in M_i with length (E) > b(M) and for which there exists at least one tetrahedron t in M_i such that E belongs to Lepp(t) and b(M) < longest edge (t) < length (E). Then the processing of E in the step1 of the algorithm implies the successive processing (in the same step1 of the algorithm) of a sequence of new terminal edges in the submesh M_i, and consequently both the refinement of these terminal edges and their associated terminal stars is performed in the same step1.*

Proof The result follows from the fact that the parallel algorithm makes implicit use of the 3-dimensional Lepp concept. In effect, the existence of t implies that there exists a sequence of interior edges in Lepp(t) which need to be traversed in order to reach the associated terminal edge E in M. Thus in the same step1 of the parallel algorithm, and in decreasing edge size order, each one of these edges becomes a terminal edge in M_i greater than $b(M)$ which is refined in the same step1.

Lemma 3.3 *The processing of an interface terminal edge greater than b(M) in step2 of the algorithm, in general implies the introduction of an interior terminal edge greater than b(M), whose processing in turn implies the introduction of an associated sequence of terminal-edges to be processed and refined in the next step1.*

Proof: Every time that the refinement of an interface terminal edge means the introduction of at least one interior terminal edge greater than b(M), the Lemma 3.2 applies and the result follows.

Lemma 3.4 *The refinement of the interface edges performed in step2 in the algorithm, in general means the introduction of a small number of interface terminal edge midpoints as compared with the number of points introduced in the processing of interior terminal edges in the step1 of the algorithm.*

Proof: First note that the processing and refinement of any interface terminal edge means the implicit use of a 'surface terminal edge bisection' algorithm which is only 1-directional (and not multidirectional) analogously to the Lepp 2-dimensional algorithm discussed in Section 2.1. From this remark and the use of the preceding lemmas, the results follows.

Its is worth pointing out that because this is a centralized algorithm, there is a chance of having a bottleneck on P_0. Strategies like latency tolerance and decentralization can be applied. However, the bottleneck behavior is very rare to appear, because of the geometrical features of the terminal stars bisections.

Lemma 3.5 *Given a submesh M_i, which has every terminal edge longer than b on its surface, an interface refinement will lead to many successive interior refinements.*

Proof: The lemma is directly deduced from the preceding lemmas.

Therefore, after sending a single UGI request for an interface vertex, a processor will have a considerable amount of work to do in the interior region before sending another UGI request for an interface vertex. This result is confirmed by our experiments, which show no difference in the query-latency both for different number of processors up to 64.

Finally it is worth noting that Proposition 1 is also valid in 3-dimensions, which together with Lemmas above guarantee algorithm termination.

4. PERFORMANCE EVALUATION

The experimental study was performed on 64 node CoWs with Sun Fire V120 processors. Each processor is UltraSPARC IIe 650 MHz / 512 KB cache / 1 GB mem. / 36 GB Ultra2 SCSI disk. The coordinator processor is a Sun Fire 280R, dual UltraSPARC III Cu 900 MHz / 8 MB cache / 2 GB mem. / 72 GB FC-AL disk. All nodes are connected to a 100Mbps Fast Ethernet Switch.

We have used two geometric models with different needs both for refinement and load imbalances: a simplified model for a human brain (see Figure 3) and a semiconductor (see Figure 4). The semiconductor model shows a more regular point distribution (almost all tetrahedra in the mesh have the same volume) than the human brain model.

The stopping criterion is a predefined bound b for the terminal edges. Next we present preliminary performance data from a Java prototype we have implemented. Specifically, we report wall time, average time for processor P_0 to assign global name to all interior and interface points, the average speed (i.e., number of new tetrahedra per second) and the quality of elements for different mesh sizes. These data illustrate good performance, scalability and stability for small to medium size CoWs i.e., for number of processors less than or equal to 64.

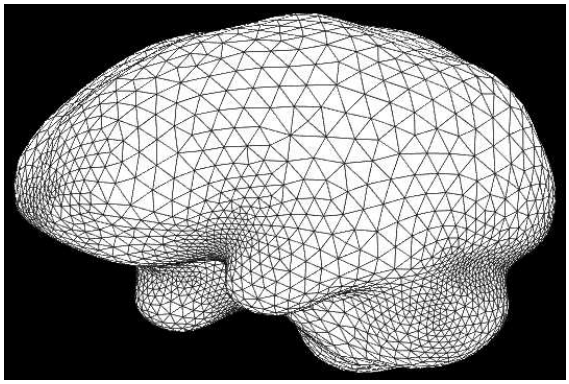

Figure 3: Surface of the tetrahedra mesh for a simplified model of a human brain generated from MRI images [16].

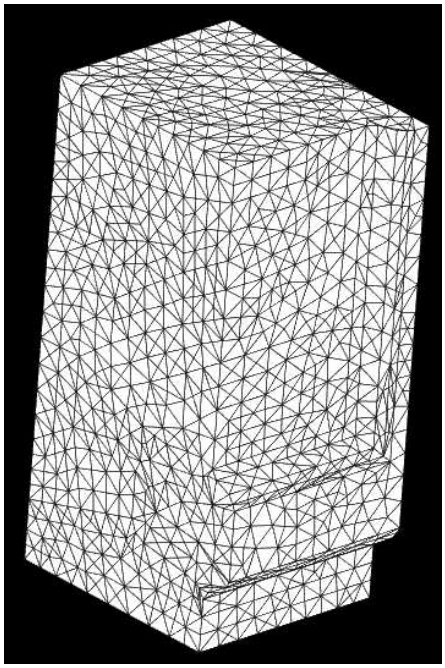

Figure 4: Surface of the tetrahedra mesh for a semiconductor.

Table 1: Time in (milliseconds) of three different mesh sizes (428K, 850K, and 2.1M elements) for the human brain model for different processor configurations and no load balancing.

Procs	*428,635*	*854,257*	*2,141,842*
8	14.3	74.6	–
16	5.54	15.0	–
24	4.32	8.39	41.7
32	3.14	8.81	38.7
40	2.85	7.34	37.3
48	1.79	3.93	22.1
56	1.79	3.81	17.6
64	1.50	4.45	18.5

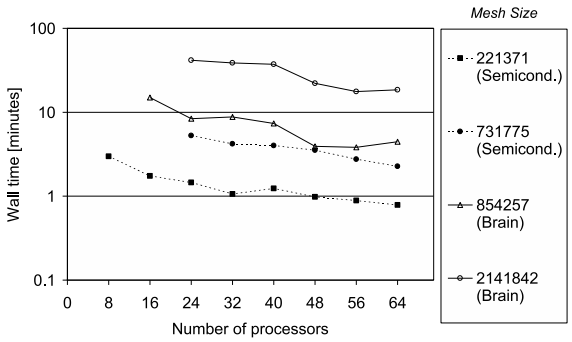

Figure 5: Wall time (in minutes) for different problem sizes and number of processors.

Table 1 presents data from the human brain which suggest that by doubling the number of processors we reduce the execution time by half. This also suggests that the coordinating processor (P_0) is not a bottleneck for up to 56 processors. However, it becomes a bottleneck (as it was expected) as we further increase this number to 64 and beyond. There is a trade-off between code simplicity and scalability, since with Java it was very easy to simplify the sequential code in order to use an additional coordinating processor for maintaining the global name space of the points. A more scalable parallel implementation of the Lepp algorithm, which uses less code re-use, is in progress. It is based on the "guided" re-partitioning idea proposed independently by Shephard's [12] and Lohner's [20] groups at RPI and GMU.

Figure 5 depicts the wall time for different problem sizes and different number of processors. The wall time clearly shows that the time decreases as we add more processors up to 56 processors. The Figure 6 shows that the method can scale to a few more processors, since the coordinating processor remains idle for about

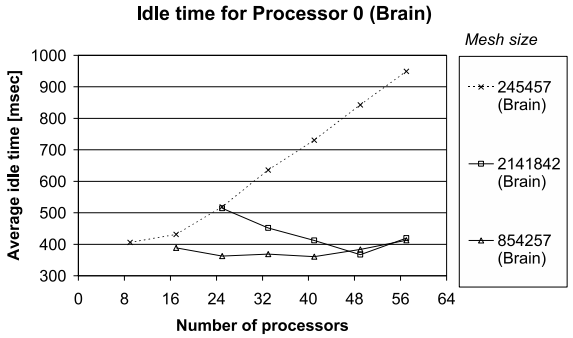

Figure 6: Idle time (in milliseconds) for the coordinating processor and the human brain model.

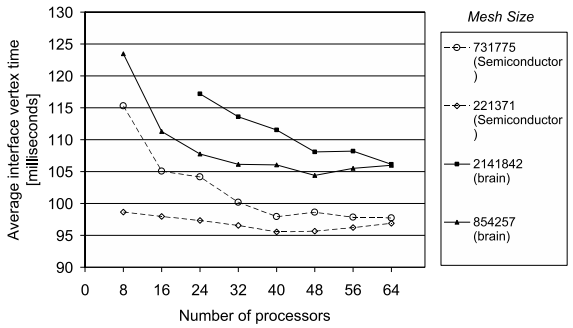

Figure 7: Average time for dealing with new / processing interface vertices.

400 milliseconds even for 64 processors. However, Table 1 shows that the wall time increases, which is due to a bottleneck in the Network Interface Card of processor P_0. This suggests that we can remove this bottleneck by using a hierarchy of coordinating processors in order to scale this method beyond to 60 processors.

Figure 7 depicts the average interface vertex time, which is the time it takes to assign a UGI for a new interface vertex. It is important that this value remain small and constant. From this graph and the equivalent graph for the Semiconductor case, we can ensure that P_0 is never overwhelmed with requests, for 64 processors.

Figure 8 shows the speed of the refinement i.e., the number of newly generated tetrahedra per second, over different problem sizes and number of processors. These graphs show the speed we can create new tetrahedra in the whole system. As we can clearly see, adding more processors to the system has a linear impact on the speed. The slope of the curve is only related with the speed of the sequential implementation for each processor. Note that our current implementation is in Java which slows down the single processor performance and thus the overall speed of the parallel implementation.

Finally, Figure 9 demonstrates (experimentally) for these two models that the method is stable i,e., with respect to the percentage of tetrahedra in quality ranges for different mesh sizes. The 3-dimensional terminal edge refinement algorithm behaves as the 2-dimensional algorithm with respect to the element quality distribution, which confirms its expected behavior: *the quality distribution is quickly displaced to the right and stabilized when the mesh has 4 times the size of the initial mesh for the brain test case, and when the mesh has 12 times the size of the initial mesh for the semiconductor case.* It is worth pointing out however that the percentage of worst tetrahedra remains quite constant, and even diminishes in the last refinement steps. More specifically, for the brain and semiconductor cases, respectively in the initial mesh there is around a 52% and a 46% of the elements in the ranges of (0.4, 0.6) and (0.3, 0.5) respectively, while that in the last refined meshes there is more than 60% of the elements in the ranges of (0.2, 0.4) for both test cases; the percentage of elements in the range (0.1) remains constant throughout the refinement steps and even diminishes in the semiconductor test case.

5. CONCLUSIONS

We have presented a practical, stable, and efficient parallel Lepp mesh generation and refinement method for tetrahedral meshes. The method works for general 3-dimensional domains and scales well up to 60 processors - size of most commonly used CoWs. The parallel algorithm is suitable for large shared memory computers. The speed of the parallel Lepp method can improve further by re-implementing the software in C or C++ and using PREMA [9] for dynamic load balancing. Our future work will focus on improving by an order of magnitude the efficiency of this method by using a simple data structure actualization function that takes full advantage of the properties of the terminal-star refinement operation. The current Java version (we used in this paper) is based on a general and unnecessarily complex function for 'general' point insertion. Also, we plan to implement an out-of-core version using the MRTS [18] runtime system in order to generate hundreds of millions of elements on relative small CoWs.

ACKNOWLEDGMENTS

This work was performed using computational facilities at the College of William and Mary which were enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund.

The comments from anonymous referees helped us to

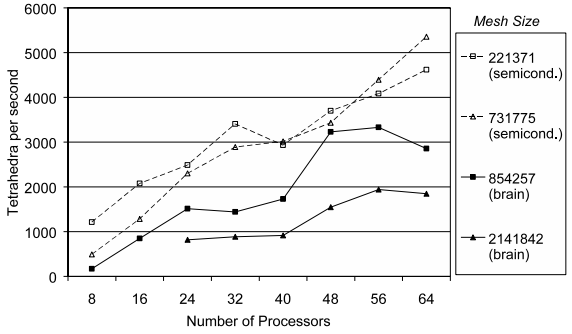

Figure 8: Number of generated tetrahedra elements per second.

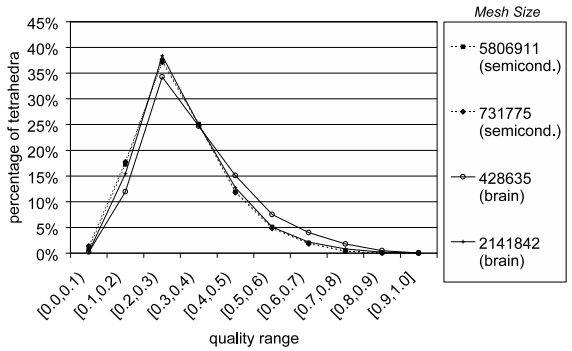

Figure 9: Quality of the mesh, for the brain and semiconductor model, measured in terms of normalized $volume/(longest\ edge)^3$ ratio.

improve the presentation of this paper.

References

[1] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali. A load balancing framework for adaptive and asynchronous applications. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):183–192, Feb. 2004.

[2] K. Barker and N. Chrisochoides. An evalaution of a framework for the dynamic load balancing of highly adaptive and irregular applications. In *Supercomputing Conference*. ACM, Nov. 2003.

[3] G. Blelloch, J. Hardwick, G. Miller, and D. Talmor. Design and implementation of a practical parallel delaunay algorithm. *Algorithmica*, 24:243–269, 1999.

[4] J. G. Castaños and J. E. Savage. The dynamic adaptation of parallel mesh-based computation. In *SIAM 7th Symposium on Parallel and Scientific Computation*, 1997.

[5] J. G. Castaños and J. E. Savage. Pared: a framework for the adaptive solution of pdes. In *8th IEEE Symposium on High Performance Distributed Computing*, 1999.

[6] D. N. N. C. P. Chew. Guaranteed–quality parallel delaunay refinement for restricted polyhedral domains. *Computational Geometry: Theory and Applications*, 28(2-3):191–215, June 2004.

[7] L. P. Chew, N. Chrisochoides, and F. Sukup. Parallel constrained Delaunay meshing. In *ASME/ASCE/SES Summer Meeting, Special Symposium on Trends in Unstructured Mesh Generation*, pages 89–96, Northwestern University, Evanston, IL, 1997.

[8] N. Chrisochoides. Multithreaded model for load balancing parallel adaptive computations. *Applied Numerical Mathematics*, 6:1–17, 1996.

[9] N. Chrisochoides, K. Barker, A. Chernikov, B. Holinka, and K. Pingali. Architecture and evaluation of a load balancing framework for adaptive and asynchronous applications. *Submitted to IEEE Trans. Parallel and Distributed Systems*, 2002.

[10] N. Chrisochoides and D. Nave. Simultaneous mesh generation and partitioning. *Mathematics and Computers in Simulation*, 54(4-5):321–339, 2000.

[11] N. Chrisochoides and D. Nave. Parallel Delaunay mesh generation kernel. *Int. J. Numer. Meth. Engng.*, 58:161–176, 2003.

[12] H. de Cougny and M. Shephard. Parallel volume meshing using face removals and hierarchical repartitioning. *Comp. Meth. Appl. Mech. Engng.*, 1999.

[13] J. Galtier and P. L. George. Prepartitioning as a way to mesh subdomains in parallel. In *Special Symposium on Trends in Unstructured Mesh Generation*. ASME/ASCE/SES, 1997.

[14] F. Gutierrez. The longest edge bisection of regular tetrahedron. In *Personal communication*, 2003.

[15] N. Hitschfeld and M. Rivara. Automatic construction of non-obtuse boundary and/or interface delaunay triangulations for control volume methods. *International Journal for Numerical Methods in Engineering*, 55:803–816, 2002.

[16] Y. Ito. Advance front mesh generator. Unpbulished Software, March 2004.

[17] M. T. Jones and P. E. Plassmann. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. In *Proceedings of the Scalable High-Performance Computing Conference*, 1994.

[18] A. Kot and N. Chrisochoides. "green" multilayered "smart" memory management system. *International Scientific Journal of Computing*, Appear 2004.

[19] L. Linardakis and N. Chrisochoides. Delaunay decoupling method for parallel guaranteed quality planar mesh generation. *Submited to SIAM Journal for Scientific Computing*, 2003.

[20] R. Lohner and J. Cebral. Parallel advancing front grid generation. *International Journal for Numerical Methods in Engineering*, 28:2889–2906, 1989.

[21] R. Lohner and J. Cebral. Parallel advancing front grid generation. In *International Meshing Roundtable*. Sandia National Labs, 1999.

[22] N. H. M. C. Rivara and R. B. Simpson. Terminal edges Delaunay (small angle based) algorithm for the quality triangulation problem. *Computer-Aided Design*, 33:263–277, 2001.

[23] M.C.Rivara. Design and data structure of fully adaptive multigrid, finite element software. *ACM Transactions on Mathematical Software*, 10:242–264, 1984.

[24] J. K. N. Hitschfeld, L. Villablanca and M. Rivara. Improving the quality of meshes for the simulation of semiconductor devices using lepp-based algorithms. *to appear. International Journal for Numerical Methods in Engineering*, 2003.

[25] K. L. L. R. B. M. N. Nambiar, R. Valera and D. Amil. An algorithm for adaptive refinement of triangular finite element meshes. *International Journal for Numerical Methods in Engineering*, 36:499–509, 1993.

[26] D. Nave, N. Chrisochoides, and L. P. Chew. Guaranteed–quality parallel Delaunay refinement for restricted polyhedral domains. In *Proceedings of the Eighteenth Annual Symposium on Computational Geometry*, pages 135–144, 2002.

[27] N. H. R.B. Simpson and M. Rivara. Approximate quality mesh generation. *Engineering with computers*, 17:287–298, 2001.

[28] M. C. Rivara. Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *International Journal for Numerical Methods in Engineering*, 20:745–756, 1984.

[29] M. C. Rivara. Selective refinement/derefinement algorithms for sequences of nested triangulations. *International Journal for Numerical Methods in Engineering*, 28:2889–2906, 1989.

[30] M. C. Rivara. New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. *International Journal for Numerical Methods in Engineering*, 40:3313–3324, 1997.

[31] M. C. Rivara and C. Levin. A 3d refinement algorithm for adaptive and multigrid techniques. *Communications in Applied Numerical Methods*, 8:281–290, 1992.

[32] M. C. Rivara and M. Palma. New lepp algorithms for quality polygon and volume triangulation: Implementation issues and practical behavior. In *In Trends unstructured mesh generationi Eds: S. A. Cannan . Saigal, AMD,*, volume 220, pages 1–8, 1997.

[33] P. S. S. R. V. N. S. N. Muthukrishnan and K. L. Lawrence. Simple algorithm for adaptative refinement of three-dimensionalfinite element tetrahedral meshes. *AIAA Journal*, 33:928–932, 1995.

[34] R. Said, N. Weatherill, K. Morgan, and N. Verhoeven. Distributed parallel delaunay mesh generation. *Comp. Methods Appl. Mech. Engrg.*, 177:109–125, 1999.

[35] D. A. Spielman, S.-H. Teng, and A. Üngör. Parallel Delaunay refinement: Algorithms and analyses. In *Proceedings of the Eleventh International Meshing Roundtable*, pages 205–217, 2001.

[36] J. P. T. Okusanya. 3D parallel unstructured mesh generation,. In *Trends in Unstructured Mesh Generation, 1997, pp. 109–116.*, 1997.

[37] S.-H. Teng. Personal communication, February 2004.

[38] R. Williams. *Adaptive parallel meshes with complex geometry.* Numerical Grid Generation in Computational Fluid Dynamics and Related Fields, 1991.